Accelerating Analytical Processing in MVCC using Fine-Granular High-Frequency Virtual Snapshotting

Ankur Sharma Felix Martin Schuhknecht Jens Dittrich

Big Data Analytics Group
Saarland University
first.last@bigdata.uni-saarland.de

ABSTRACT

Efficient transaction management is a delicate task. As systems face transactions of inherently different types, ranging from point updates to long-running analytical queries, it is hard to satisfy their requirements with a single execution engine. Unfortunately, most systems rely on such a design that implements its parallelism using multi-version concurrency control. While MVCC parallelizes short-running OLTP transactions well, it struggles in the presence of mixed workloads containing long-running OLAP queries, as scans have to work their way through vast amounts of versioned data. To overcome this problem, we reintroduce the concept of hybrid processing and combine it with state-of-the-art MVCC: OLAP queries are outsourced to run on separate virtual snapshots while OLTP transactions run on the most recent version of the database. Inside both execution engines, we still apply MVCC.

The most significant challenge of a hybrid approach is to generate the snapshots at a high frequency. Previous approaches heavily suffered from the high cost of snapshot creation. In our approach termed AnKer, we follow the current trend of co-designing underlying system components and the DBMS, to overcome the restrictions of the OS by introducing a custom system call `vm_snapshot`. It allows fine-granular snapshot creation that is orders of magnitudes faster than state-of-the-art approaches. Our experimental evaluation on an HTAP workload based on TPC-C transactions and OLAP queries show that our snapshotting mechanism is more than a factor of 100x faster than fork-based snapshotting and that the latency of OLAP queries is up to a factor of 4x lower than MVCC in a single execution engine. Besides, our approach enables a higher OLTP throughput than all state-of-the-art methods.

CCS CONCEPTS

• **Information systems → Main memory engines**;

ACM Reference Format:
Ankur Sharma, Felix Martin Schuhknecht, and Jens Dittrich. 2018. Accelerating Analytical Processing in MVCC using Fine-Granular High-Frequency Virtual Snapshotting. In *SIGMOD'18: 2018 International Conference on Management of Data, June 10–15, 2018, Houston, TX, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3183713.3196904

SIGMOD'18, June 10–15, 2018, Houston, TX, USA

ACM ISBN 978-1-4503-4703-7/18/06.
https://doi.org/10.1145/3183713.3196904

1 INTRODUCTION

Realizing fast concurrent transactional processing is a desirable but challenging design goal. A concurrency control technique is required to fully utilize the massive amount of hardware parallelization that is nowadays available even in commodity servers.

Interestingly, a large number of database systems, including major players like PostgreSQL [18], Microsoft Hekaton [6], SAP HANA [7], HyPer [16], MemSQL [1], MySQL [2], NuoDB [3], and Peloton [4] currently implement a form of multi-version concurrency control (MVCC) [5, 14, 24] to manage their transactions. It allows for a high degree of parallelism as readers do not block writers. The core principle is straightforward: if a tuple is updated, a new physical version of this tuple is created and stored alongside the old one in a version chain, such that the old version is still available for readers that are still allowed to see the older version. Timestamps ensure that transactions access only the most recent version that existed when they entered the system.

1.1 Limitations of MVCC

In MVCC implementations that rely on a single execution engine, all transactions, no matter whether they are short running OLTP transactions or scan-heavy OLAP queries, are treated equally and are executed on the same (versioned) database. While this form of processing unifies the way of transaction management, it also has unpleasant downsides under HTAP workloads: First and foremost, scan-heavy OLAP queries heavily suffer when they have to deal with a large number of version chains [16]. During a scan, version chains must be traversed to locate the most recent version of each item that is visible to the transaction. It involves expensive timestamp comparisons as well as random accesses when going through the version chains. As column scans typically take significantly more time than short-running transactions, which touch only a few entries, a large number of OLTP transactions can perform updates in parallel to create such version chains. Apart from this, these version chains must be garbage collected from time to time to remove versions that are not visible to any transaction in the system. Garbage collection is typically done by a separate thread, which frequently traverses these chains to locate and delete outdated versions [12, 25, 26]. This thread has to be managed and synchronized with the transaction processing, utilizing precious system resources.

HTAP workload, consisting of transactions of inherently different nature, does not fit the uniform processing in a single execution engine, which treats all incoming transactions in the same way. Unfortunately, many state-of-the-art MVCC systems [1, 2, 4, 6, 16, 18] implement some variant of such a processing model.

1.2 Hybrid Processing

But why exactly do these systems rely on such a processing model, although it does not fit the faced workload? Why they do not implement *hybrid processing*, which classifies queries based on the type and executes them in separation?

To answer these questions, let us look at the development of the prominent HyPer [10, 16] system. Early versions of HyPer implemented hybrid processing [10, 15]: the queries were classified into the categories OLTP and OLAP and consequently executed on separate representations of the database. The short running modifying OLTP transactions were executed on the most recent version of the data while long-running OLAP queries were outsourced to run on *snapshots*. These snapshots were created from time to time on the up-to-date version of the database.

While this concept mapped the mixed workload to the processing system in a natural way, the developers faced a severe problem: the creation of snapshots turned out to be too expensive [16]. To snapshot, HyPer utilized the `fork` system call. This system call creates a child process that shares its virtual memory with the parent process. Both processes perform copy-on-write to keep changes locally, thus implementing virtual snapshotting. While this principle is cheaper than physical snapshotting, which eagerly creates a deep copy of the data, forking a process has a considerable overhead of replicating the entire virtual memory allocated by the parent process. Thus, the developers decided to move away from hybrid processing to a model with a single execution engine, relying entirely on MVCC in their current version [16].

1.3 Challenges

Despite the challenges one has to face when implementing a hybrid model, we believe it is the right choice. Matching the processing system to the workload is crucial for performance. This is the goal of our processing concept termed *AnKer*, which we will propose in the following. Still, to do so, we have to discuss two problems: **(a)** Obviously, MVCC is the state-of-the-art concurrency control mechanism in main-memory systems. We intend to apply it as well to parallelize transaction processing *within* our execution engines. But how to combine state-of-the-art MVCC with a hybrid processing model? **(b)** State-of-the-art snapshotting mechanisms are not capable of powering a hybrid processing model. How to realize a fast snapshotting mechanism, that allows the creation of snapshots at a high frequency and with high flexibility?

2 BACKGROUND

Classical systems implement MVCC in a single execution engine, where all queries are treated equally and executed on the same versioned database. In contrast to that, AnKer extends the capabilities of MVCC by reintroducing the concept of *hybrid processing*, where incoming OLTP transactions and OLAP queries are treated independently. By this, we can utilize the advantages of MVCC *while* avoiding its downsides.

2.1 MVCC in Hybrid Processing

The concept of hybrid processing works as follows: based on the classification, we separate the short-running OLTP transactions from the long-running (read-only) OLAP queries. Conceptually, the modifying OLTP transactions run concurrently on the most recent version of the database and build up version chains as in classical MVCC. In parallel, we outsource the read-only OLAP queries to run on separate (read-only) snapshots of the versioned database. These snapshots are created at a very high frequency to ensure freshness. Thus, instead of dealing with a single representation of the database that suffers from a large number of long version chains, we maintain a most recent version in an OLTP execution engine alongside with a set of snapshots, which are present in the OLAP execution engine. Naturally, each of the representations contains fewer and shorter version chains, which primarily reduces the main problem described in Section 1.1. Apart from that, using snapshots has the pleasant side-effect that the garbage collection of version chains becomes extremely simple: We remove the version chains automatically with the deletion of the corresponding snapshot if it is not visible to any transaction in the system. Other systems like PostgreSQL have to rely on a fine-granular garbage collection mechanism for shortening the version-chains, requiring precious resources. By using snapshotting, we can solve the problem of complex garbage collection techniques implicitly.

2.2 High-Frequency Snapshotting

With the high-level design of the hybrid processing model at hand, the question remains how to realize efficient snapshotting. The approach stands and falls with the ability to generate snapshots at a *very high frequency* to ensure that transactions running on the snapshots have to deal only with few and short version chains. In this regard, previous approaches that relied on snapshotting suffered under the expensive snapshot creation phase and consequently moved away from snapshotting. As mentioned, early versions of HyPer [10], which also used a hybrid processing model, created virtual snapshots utilizing the system call `fork`. This call is used to spawn child processes which share their entire virtual memory with the parent process. The copy-on-write, which is carried out by the operating system on the level of memory pages ensures that changes remain local in the associated process. While this mechanism naturally implements a form of snapshotting, process forking is expensive. Thus, it is not an option for our case as we require a more lightweight snapshotting mechanism.

In a recent publication on the *rewiring* [19] of virtual memory, the authors also looked into the case of snapshot creation. With rewiring, they can manipulate the mapping from virtual to physical memory pages at runtime in userspace. In [19], the authors used this technique to snapshot an existing virtual memory area v_1, which maps to a physical memory area p_1, by manually establishing a mapping of a new virtual memory area v_2 to p_1. While this approach is already significantly faster than using `fork`, it is still not optimal as the mapping must be reconstructed using `mmap` page-wise n ithe worst case — a costly process for large mappings.

Unfortunately, all the existing solutions are not sufficient for our requirements on snapshot creation speed. Therefore, AnKer implements a more sophisticated form of virtual snapshotting. We do not limit ourselves by using the given general purpose system calls. Instead, we introduce our custom system call termed `vm_snapshot` and integrate the concept of rewiring [19] directly into the Linux kernel. Such a co-design of underlying system components and the DBMS has been demonstrated successfully in recent publications concerning both operating system [9, 13] and hardware [17, 21] customizations, as it enables a whole new level of optimization opportunities. Using our call, we can essentially snapshot arbitrary

virtual memory areas within a single process at any point in time. The virtual snapshots share their physical memory until a write to a virtual page happens which allows us to create snapshots with a small memory footprint, allowing us to build them at a high frequency without much memory overhead. Consequently, the individual snapshots contain few and short version chains and enable efficient scans.

2.3 Structure & Contributions

Before we start with a detailed presentation of the system and the individual components, let us outline the contributions we make:
(I) We present AnKer, a hybrid storage model that is able to execute scan-heavy OLAP queries on a consistent snapshot while processing short running transactions over the most recent version of the database. We extend this model to redesign HyPer's MVCC engine [16] as an example, to show the benefits of a hybrid processing model over conventional MVCC implementations. We also show that the changes to the original implementation are minimal and can be easily adopted to other main-memory MVCC systems [11, 20, 23].
(II) We realize the snapshots in form of *virtual snapshots* and heavily accelerate the snapshotting process by introducing a *custom system call* termed `vm_snapshot` to the Linux kernel. This call directly manipulates the virtual memory subsystem of the OS and allows for a significantly higher snapshotting frequency than state-of-the-art techniques. We demonstrate the capabilities of `vm_snapshot` in a set of micro-benchmarks and compare it against the existing physical and virtual snapshotting methods.
(III) We create snapshots on the *granularity of a column*, instead of snapshotting the entire table or database as a whole which is possible due to the flexibility of our custom system call `vm_snapshot`. Therefore, we can limit the snapshotting effort to those columns, which are accessed by the transactions.
(IV) We create *snapshots of versioned columns*. To create a snapshot, the current column as well as the timestamp information is virtually snapshotted using our custom system call `vm_snapshot`, and the current version chains are handed over. Running transactions can still access all required versions from the fresh snapshot. As the snapshot is read-only, all further updates happen to the up-to-date column, creating new version chains. As a side-effect, we avoid any expensive garbage collection as dropping an old snapshot drops all old version chains with it.
(V) We perform an extensive experimental evaluation of AnKer. We compare the hybrid processing model utilizing our system call `vm_snapshot` with a fork-based snapshotting approach. Additionally, we compare the hybrid models with a single execution engine under full serializability, snapshot isolation, and read uncommitted guarantees, executing mixed HTAP workloads based on TPC-C transactions and configurable OLAP queries. Our prototype implementation of the AnKer concept can be configured to support both a hybrid and a single execution engine (by disabling snapshotting) as well as the required isolation levels. We show that our approach offers faster snapshotting, lower OLAP latency and higher OLTP transaction throughput than the counterparts under mixed workloads.

The paper has the following structure: In Section 3, we describe the hybrid design of AnKer and motivate it with the problems of state-of-the-art MVCC approaches. As the hybrid execution engine requires a fast snapshotting mechanism, we discuss the currently available snapshotting techniques to understand their strengths and weaknesses in Section 4. In Section 5, we propose our snapshotting method based on our custom system call `vm_snapshot`. Finally, in Section 6, we evaluate AnKer in different configurations and show the superiority of hybrid processing using `vm_snapshot`.

3 ANKER

As outlined, the central component of AnKer is a hybrid processing model, which separates OLTP from OLAP processing using virtual snapshotting. Both in the up-to-date representation of the data as well as in the snapshots, we want to use MVCC as the concurrency control mechanism. To understand our hybrid design, let us first understand how MVCC works on a single execution engine.

3.1 Mechanisms of MVCC

To understand the mechanisms of MVCC, let us go through the individual components. Initially, the data is unversioned and present in the column. Thus, there does not exist any version chains. If a transaction updates an entry, we first store the new value locally inside the local memory of the transaction. Update are materialized in the column when the actual commit happens. Before applying the update to the in-place value, the system copies the old value to the version chain using atomic compare-and-swap instructions. We store the versions in a *newest-to-oldest* order. Other systems as, e.g., HyPer [16] rely on this order as well, as it favors younger transactions: they will find their version early on during the chain traversal. A version chain can become arbitrarily long if frequent updates to the same entry happen. Along with the version, we store a unique timestamp of the update that created the version which is necessary to ensure that the transactions that started before the (committed) update happened, do not see the new version of the entry but still the old one. Unfortunately, reading a versioned column can become arbitrarily expensive since the chain must be traversed using the comparison of timestamps to locate the proper version. In summary, if a large number of lengthy version chains is present and a transaction intends to read many entries, the version chain traversal cost becomes significant.

Besides the way of versioning the data, the guaranteed isolation level is an important aspect in MVCC. As a consequence of its design, MVCC implements snapshot isolation guarantees by default. During its lifetime, a transaction T sees the committed state of the database, that was present at T's start time. The updates of newer transactions, which committed during T's lifetime, are not seen by T. Write-write conflicts are detected at commit time: if T wants to write to an entry, to which a newer committed transaction already wrote, T aborts. Still, under snapshot isolation, so-called write-skew [8] anomalies are possible. Fortunately, MVCC can be extended to support full serializability [16, 22]. To do so, we extend the commit phase of a transaction with additional checks. If a transaction T wants to commit, it validates its read-set by inspecting if any other transaction, that committed during T's lifetime changed an entry in a way that would have influenced T's result. If this is the case, T has to abort as its execution was based on stale reads. To perform the validation, we adopt the efficient approach applied in HyPer [16], which is based on precision locking [24], a variant of predicate locking. Essentially, the system tracks the predicate

ranges on which the transaction filtered the query result. During validation, we check whether any write of any recently committed transaction intersects with the predicate ranges. If an intersection is identified, the transaction aborts.

3.2 Hybrid MVCC

To overcome the limitations of MVCC implementations mentioned above, we realize a hybrid execution engine in AnKer. Two engines are present side by side: one engine is responsible for the concurrent processing of short-running transactions (termed *OLTP execution engine* in the following), while the other one can perform long-running read-only transactions in parallel (termed *OLAP execution engine* from here on). Incoming transactions are marked as either an OLTP transaction or an OLAP query and sent to the respective engine for processing. The challenge is to combine the concept of hybrid processing with MVCC. Let us look at the engines in detail in the case of an example depicted in Figure 1. In the example, we use the following set of operations:

$w_i(v)$	write to row i the given value v
$r_i() \rightarrow v$	read from row i and return the read value as v
$sum() \rightarrow r$	sum up all column values and return the result as r
$avg() \rightarrow r$	average all column values and return the result as r

Step ①: For the following discussion, we assume that our table consists of a single column C of 6 rows, identified by rows 0 to 5, which all contain the value 0 in the beginning. This column C is located in the OLTP execution engine and is the up-to-date representation of the column. Since there are no snapshots present yet, the OLAP execution engine virtually does not exist.

Step ②: Two OLTP transactions T_1 and T_2 arrive and intend to perform a set of writes. The first write $w_5(1)$ of T_1 intends to update at row 5 the value 0 with the new value 1. However, instead of replacing the old value in the column with the new value, we store the new value locally inside the transaction T_1 and keep the column untouched as long as the transaction does not commit. In the same fashion, the remaining write $w_1(2)$ of T_1 as well as the write $w_3(3)$ of T_2 are performed only locally inside their respective transactions. Note that all three written values are uncommitted so far and are visible to the transactions that completed the individual writes.

Step ③: Let us now assume that T_1 commits while T_2 intentionally aborts. The commit of T_1 now replaces the old value 0 with the new value 1 in column C at row 5. Of course, the old value 0 is not discarded but stored in a newly created version chain for that row. Similarly, at row 1 the old value 0 is replaced by the new value 2, and the old value stored in the version chain. Note that we implement a timestamp mechanism (logging both the start and end time of a transactions commit phase) to ensure that both writes of T_1 become visible atomically to other transactions. As no other transactions modified row 1 and 5 during the lifetime of T_1, the commit succeeds and satisfies full serializability, that we guarantee for all transactions. In contrast to that, the abort of T_2 discards merely the local change of row 3. This strategy with no rollback makes aborts cheap.

Step ④: An OLAP query Q_1 arrives, which intends to scan and sum up the values of all rows of the column, denoted by $sum()$. To execute Q_1, first, a snapshot of column C is taken utilizing our custom system call (which we described in Section 5 in detail), resulting in a (virtual) duplicate of the column denoted as C'. It

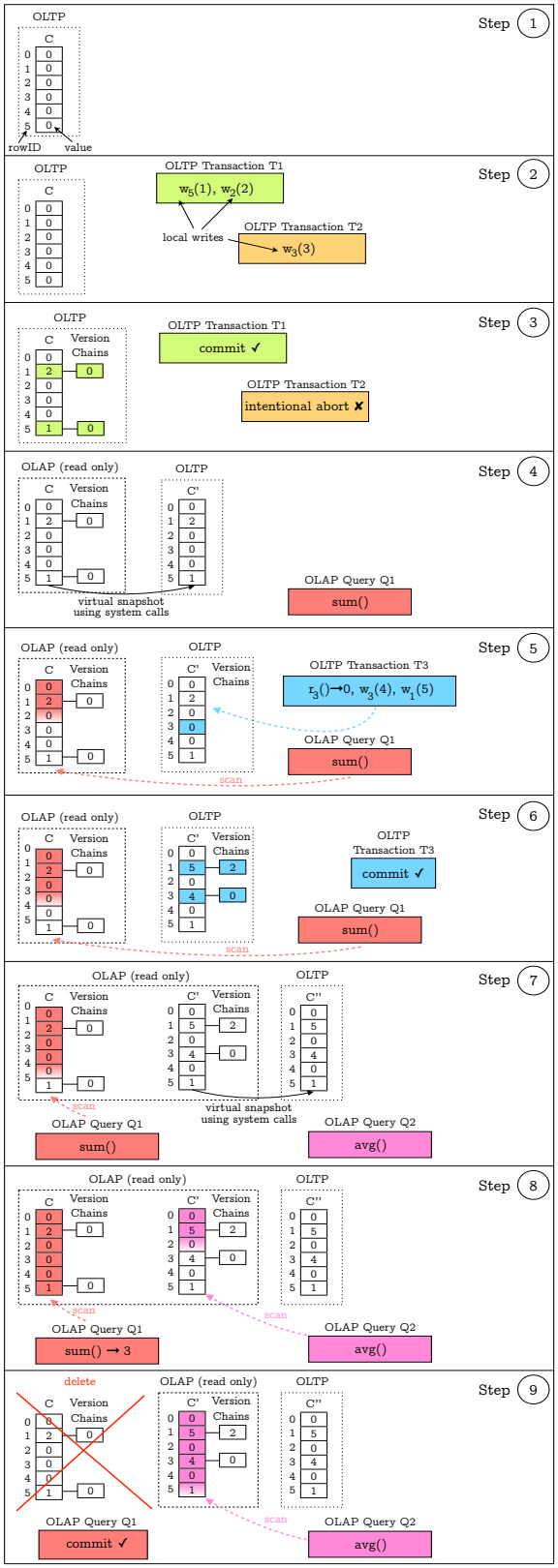


Figure 1: Hybrid processing in AnKer.

is essential to understand that this duplicate C' will become the most recent version of the column in the OLTP engine. The "old" column C along with its build-up version chains is logically moved to the OLAP engine and becomes read-only.

Step ⑤: Another OLTP transaction T_3 arrives, that intends to perform a read $r_3()$ followed by the two writes $w_3(4)$ and $w_1(5)$. The read $r_3()$ is performed by accessing the current value of row 3 of the representation in the OLTP view, resulting in $r_3() \rightarrow 0$. The two successive writes are stored locally inside T_3 and are not visible for other transactions. In parallel to the depicted operations of T_4, our OLAP query Q_1, which sums up the column values, starts executing in the OLAP engine on C. As the snapshot is older than Q_1, it can directly scan C without inspecting the version chains.

Step ⑥: While the scan of Q_1 is running, T_3 decides to commit. This commit does not conflict with the execution of Q_1 in any way, as the Q_1 and T_3 run in different execution engines. The local write $w_3(4)$ and $w_1(5)$ are materialized in C' after moving the old version to version chain.

Step ⑦: Another OLAP query Q_2 arrives, which attempts to compute the average of the column, denoted by $avg()$ triggers the creation of a new snapshot. Again we use our system call and take a snapshot of column C' that is located in the OLTP engine, resulting in a (virtual) duplicate of the column in form of C''. The new duplicate C'' becomes the most recent representation of the column in the OLTP engine, while C' with its version chains re-labeled as the OLAP engine. Note that both C' as well as C is now present in the OLAP engine side by side, with Q_1 still running on C.

Step ⑧: The new OLAP query Q_2 starts running on the new snapshot C', while the older OLAP query Q_1 finishes its scan, returns the sum 3 and commits.

Step ⑨: The finish of Q_1 makes C obsolete, as a newer representation C' already exists and no transaction accessing C is running. Thus, we can safely delete the oldest snapshot C.

3.2.1 Snapshot Synchronization. For simplicity, in the previous example, all transactions worked solely on a single column. However, a database usually consists of several tables, containing a large number of attributes and therefore, some form of *snapshot synchronization* is necessary. In this context, snapshot synchronization means that a transaction, which accesses multiple columns, has to see all columns consistent concerning a single point in time. The system could trivially snapshot all columns of all tables for each request of the snapshot. However, this causes unnecessary overhead as we might access only a small subset of the attributes. Therefore, in AnKer, we implement a lazy snapshot materialization approach. The system logs the snapshot-timestamp along with the list columns used by the snapshot for each snapshot request. The actual snapshot materialization happens for each column if an OLTP transaction or an OLAP query comes in, which accesses the columns requested by the snapshot. This lazy strategy ensures that columns, that are never used by OLAP queries or are never updated by transactions are also never materialized in snapshots.

3.2.2 Snapshot Consistency. In the previous example, we created a new snapshot for each OLAP query. When this happens, and the previously described access triggers the actual materialization of the snapshot using our system call, we have to ensure that no other transactions modify the column while the snapshot is under creation. We ensure this using a shared lock on the column, which must be acquired by any transaction which performs an installation of updates to the respective column during the commit phase. When materializing a snapshot, an exclusive lock must be acquired. To grant an exclusive lock, the system blocks all further requests for the shared lock and the snapshot can be materialized once all already-acquired shared locks are released.

4 STATE-OF-THE-ART SNAPSHOTTING

As stated before, our hybrid processing model stands and falls with an efficient snapshot creation mechanism. Only if we can create them at a high frequency without penalizing the system, we get up-to-date snapshots with short version chains. There exist different techniques to implement such a snapshotting mechanism, including *physical* and *virtual* techniques. While the former ones create costly physical copies of the entire memory, the latter ones lazily separate snapshots only for modified memory pages. Let us now look at the state-of-the-art techniques in detail to understand why they do not suffice our needs and why we have to introduce an entirely new snapshotting mechanism in AnKer.

4.1 Physical Snapshotting

The most simple approach is *physical snapshotting*, where a deep physical copy of the database is created. On this physical copy, the reading queries can then run in isolation, while the modifying transactions update the original version. The granularity of snapshotting is an important design decision. It is possible to snapshot the entire database, a table, or just a set of columns. This way of snapshotting represents the *eager* way of doing it — at the time of snapshot creation, the snapshot and the source are entirely separated from each other. As a consequence, any modification to the source is not carried through to the snapshot.

Physical snapshotting is straightforward and is easy to apply. However, its effectiveness is directly bound to the amount of data that is updated on the source. If only a portion of the data is updated, the full physical separation of the snapshot and the source is unnecessary and just adds overhead to the snapshotting cost.

4.2 Virtual Snapshotting

Virtual Snapshotting overcomes this problem by following the *lazy* approach. The idea of virtual snapshotting is that initially the snapshot and the source are not separated physically. Instead, the separation happens lazily only for those memory pages that are modified. As we will see, there are multiple ways to perform this separation using virtual memory. To understand them, let us first go through some of the high-level concepts of the virtual memory subsystem of *Linux* (kernel 4.8).

4.2.1 Virtual vs Physical Memory. By default, the user perspective on memory is simple — only virtual memory is visible. To allocate a consecutive virtual memory area b of size s the system call `mmap` is used. For instance, the general purpose memory allocator `malloc` from the GNU C library internally uses `mmap` to claim large chunks of virtual memory from the operating system. The layer of physical memory is completely hidden and transparently managed by the operating system. After allocating the virtual memory area, the user can start accessing the memory area, e.g., via $b[i] = 42$. Apparently, the user perspective is relatively simple. He does not have to distinguish between memory types at all. In comparison, the kernel perspective is significantly more complicated.

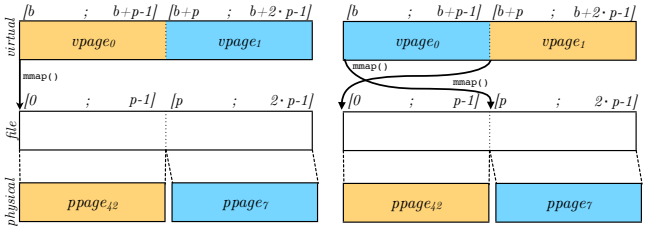


Figure 2: Visualization of rewiring as shown in [19]. The start address of the virtual memory area is denoted as b and the page size as p. A consecutive virtual memory area of two pages is mapped to a main-memory file, which is transparently mapped to two potentially scattered physical pages (left part). The system call `mmap` can be used to manipulate the mapping at runtime (right part).

First of all, the previously described call to `mmap`, which allocates a consecutive virtual memory area, does not trigger the allocation of physical memory right away. Instead, the call only creates a so-called `vm_area_struct` (*VMA*), that contains all relevant information to *describe* this virtual memory area. For instance, it stores that the size of the area is s and that the start address is b. Thus, the set of all VMAs of a process defines which areas of the virtual address space are currently *reserved*. Note that a single VMA can describe a memory area spanning over multiple pages. As an example, in Figure 3 we visualize two VMAs. They describe the virtual memory areas starting at address b (spanning over four pages) and c (spanning over three pages). In between the two memory areas is an unallocated memory area of size two pages. Besides the VMAs, there exists a *page table* within each process. A page table entry (*PTE*) that contains the actual mapping from a single virtual to a physical page is inserted after the first access to a virtual page, based on the information stored in the corresponding VMA. The example in Figure 3 shows the state of the page table after four accesses to four different pages. As we can see, there is one PTE per accessed page in the page table.

4.2.2 Fork-based Snapshotting. With the distinction between the different memory types and the separation of VMAs and PTEs in mind, we are now able to understand the most fundamental form of virtual snapshotting: fork-based snapshotting [10]. It exploits the system call `fork`, which creates a child process of the calling parent process. This child process gets a copy of all VMAs and PTEs of the parent. In particular, this means that after a fork, the allocated virtual memory of the child and the parent share the same physical memory. Only a write[1] to a page of child or parent triggers the actual physical separation of that page in the two processes (called *copy-on-write* or *COW*). This concept can be exploited to implement a form of snapshotting. If the source resides in one process, one can merely fork it to create a snapshot. Any modification to the source in the parent process is not visible in the child process. As mentioned in Section 1.2, early versions of HyPer that implemented hybrid processing utilized `fork`.

4.2.3 Rewired Snapshotting. While fork-based snapshotting has the convenient advantage, that the snapshotting mechanism is handled by the operating system in a transparent manner, it has two significant disadvantages. First, it requires the spawning and management of several processes at a time. Second, it always snapshots all allocated memory of the process, i.e., it cannot be used to snapshot a subset of the data. Both problems can be addressed using the technique of rewiring as described in [19].

To understand rewiring, let us again look at the mapping from virtual to physical memory as described in Section 4.2.1. This mapping is by default both *hidden* from the user as well as *static*, as the user sees only virtual memory by default. The authors of [19] manage to reintroduce physical memory to userspace in the form of so-called *main-memory files*. A main-memory file has the same properties as a file on disk, except that volatile main-memory instead of disk pages back it. It can be mapped to a virtual memory region using the system call `mmap` and accessed through it. As it is possible to manipulate the mapping from virtual memory to the main-memory files using `mmap` and main-memory files are internally backed by physical memory; they can establish a transitive mapping from virtual to physical memory. At any time, this mapping can be modified using `mmap`. Using rewiring memory, it is possible to establish a mapping that is both *visible* and *modifiable* in userspace. To understand the concept, consider the example in Figure 2 from the original paper [19] that swaps the content of two pages. On the left side, two virtual memory pages of size p starting at virtual address b are mapped to a main-memory file at offset 0. Since the main-memory file is transparently backed by the two physical pages $ppage_4 2$ and $ppage_7$, the mappings $vpage_0 \rightarrow ppage_{42}$ and $vpage_1 \rightarrow ppage_7$ have been established. Using `mmap`, it is now easily possible perform the swapping by mapping $vpage_0$ to file offset p and $vpage_1$ to file offset 0. This changes the physical pages that are backing the virtual pages, resulting in a change of content.

In rewired snapshotting, the authors of [19] utilize this modifiable mapping. Let us assume there is a virtual memory area b, on which a snapshot should be created. To snapshot, the authors simply allocate a new virtual memory area c and `mmap` (or rewire) it to the file, which represents the physical memory, in the same way, as b. Consequently, b and c share the same physical pages. If now a write to a page of b is happening, the separation of the snapshot and original version must be performed manually on that page, before the write can be carried out. In the first place, the write must be detected. After detection, an unused page is claimed from the file (which serves as the pool for free pages), the page content is copied over, the write is performed, and b is rewired to map to the new page. By this, it is possible to mimic the behavior of `fork` while staying within a single process. Further, the technique offers the flexibility of snapshotting only a fraction of the data. However, rebuilding the mapping can also be quite expensive as we will see.

4.3 Reevaluating the State-of-the-Art

As we have discussed the different state-of-the-art methods of physical and virtual snapshotting that are present, let us now try to understand their strengths and limitations. This analysis will point us directly to the requirements we have on our custom system call, that we will use in AnKer to power snapshotting. In the experiment we are going to conduct in Section 4.3.2, we evaluate the time to *create a snapshot* in the sense of establishing a separate view on the data. While for physical snapshotting, this means creating a deep physical copy of the data, for virtual snapshotting, it does not

[1] Assuming the virtual memory area written to is private (`MAP_PRIVATE`).

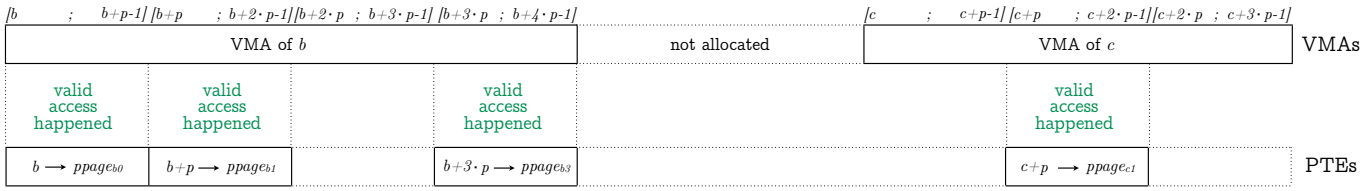


Figure 3: Visualization of the relationship between VMAs and PTEs. The VMAs store the information about the currently allocated virtual memory areas alongside with all necessary meta-information.

trigger any physical copy of the data. Still, virtual snapshotting has to perform a certain amount of work as we will see. We will perform the experiment as a stand-alone micro-benchmark to focus entirely on the snapshotting costs and to avoid interference with other components, that are present in our prototype of the AnKer concept. We use a table with $n = 50$ columns, stored in a columnar fashion, where each column has a size of 200MB. The question remains which page size to use. To make snapshotting as efficient as possible, we want to back our memory with pages *as small as available*. This ensures that the overhead of copy-on-write on the level of page granularity is minimal. Consider the case where our 200MB column is either backed by 100 huge pages or 51,200 small pages. In the former case, 100 writes would cause a COW of the entire column (200MB) in the worst case, resulting in a full physical separation of the snapshotted column and the base column. In the latter case, 100 writes would trigger COW of at most 100 small pages (400KB), physically separating only 0.2% of the snapshotted column from the base column.

4.3.1 System Setup. Before the start of the evaluation, let us look at the setup. We perform all of the following experimental evaluations on a server consisting of two quad-core Intel Xeon E5-2407 running at 2.2 GHz. The CPU does neither support hyper-threading nor turbo mode. The sizes of the L1 and L2 caches are 32KB and 256KB, respectively, whereas the shared L3 cache has a capacity of 10MB. The processor can cache 64 entries in the fast first-level data-TLB for virtual to physical 4KB page address translations. In a slower second-level TLB, 512 translations can be stored. In total, the system is equipped with 48GB of main memory, divided into two NUMA regions of 24GB each. For the upcoming micro-benchmarks of this Section, we deactivate one CPU and the attached NUMA region to stay local on one socket. For the experimental evaluation in Section 6, we use both sockets. The operating system is a 64-bit version of Debian 8.16 with our customized Linux kernel (version 4.8.17), that has been extended with our `vm_snapshot` system call. The codebase is written in C++ and compiled using g++ 6.3.0 with optimization level O3.

4.3.2 Creating a Snapshot. To simulate snapshotting on a subset of the data, we create a snapshot on the first p columns of the table T. Let us precisely define how the individual snapshotting techniques behave in this situation:

(a) Physical: to create a snapshot of p columns of table T, we allocate a fresh virtual memory area S of size $p \cdot l$ pages, where l denote the number of pages per column. Then, we copy the content of p columns of T into S using `memcpy`. S represents the snapshot.

(b) Fork-based: to create a snapshot of p columns of table T, we create a copy of the process containing table T using `fork`. Independent of p, this snapshots the entire table. The first p columns of table T' contained in the child process represent the snapshot. The virtual memory areas representing T and T' are declared as *private*, such that writes to one area are isolated from the other area.

(c) Rewiring: to create a snapshot of p columns of table T, we first have to inspect by how many VMAs each column is actually described. As a VMA describes the characteristic properties of a consecutive virtual memory region, it is possible that a column is represented by only a single VMA (best case), by one VMA per page (worst case), or anything in between. The more writes happened to a column and the more copy-on-writes were performed, the more VMAs a column is backed by. Eventually, every page is described by its individual VMA. To create the snapshot, we first allocate a fresh virtual memory area S of size $p \cdot l$ pages, where l denote the number of pages per column. For each VMA that is backing a portion of the p columns in T, we now rewire the corresponding part of S to the same file offset. Additionally, we use the system call `mprotect` to set the protection of S to read-only. This is necessary to detect the first write to a page to perform a manual copy-on-write. S represents the snapshot.

Table 1: Creating a snapshot using state-of-the-art techniques. We vary the number of columns on which we snapshot. For rewiring, the number of modified pages influences the runtime. Thus, we show the snapshotting cost after 0, 500, 5000, and 50000 pages were modified per column.

Method	Pages Modified per Column	1 Col [ms]	25 Col [ms]	50 Col [ms]
Physical	–	108.09	2693.69	5382.87
Fork-based	–	108.28	108.28	108.28
Rewiring	0	0.02	0.39	7.72
Rewiring	500	1.22	30.90	61.87
Rewiring	5000	14.17	352.15	712.96
Rewiring	50000	169.28	4210.17	8459.67

Table 1 shows the results. We vary the number of columns to snapshot p from 1 column (2% of the table) over 25 columns (50% of the table) to 50 columns (100% of the table) and show the runtime in ms to create the snapshot. For rewiring, we vary the pages that have been modified (by writing the first 8B of the page) before the snapshot is taken, as it influences the runtime. We test the case where no write has happened, and a single VMA backs each column. Further, we measure the snapshotting cost after 500 pages, 5000 pages, and 50000 pages have been modified. These number of writes lead to 995, 9483, and 51177 number of VMAs backing a column. First of all, we can see that physical snapshotting is quite expensive, as it creates a deep copy of the columns already at snapshot creation time. As expected, we can observe a linearly increasing cost with the number of columns to snapshot. In contrast to that, fork-based snapshotting is independent of the number of requested columns, as it snapshots the entire process with the whole table in any case.

When snapshotting 50% of the table, fork-based snapshotting is over an order of magnitude faster than physical snapshotting, as it duplicates solely the virtual memory, consisting of the VMAs and the page table. The runtime of rewiring is highly influenced by the number of written pages, respectively the number of VMAs per column. The more VMAs we have to touch to create the snapshot, the higher the runtime. If we have as many VMAs as pages (the case after 50000 writes), the runtime of rewiring is higher than the one of physical snapshotting. However, we can also see rewiring is significantly faster than the remaining methods if fewer VMAs need to be copied. For instance, after 500 writes, rewiring is around two orders of magnitude faster for a single column and almost factor two faster for snapshotting the entire table.

4.3.3 Summary of Limitations. The performance of rewiring for snapshot creation is highly influenced by the number of VMAs per column. For every VMA, a separate `mmap` call must be carried out – a significant cost if the number of VMAs is large. Unfortunately, when using rewiring, an increase in the amount of VMAs over time is not avoidable.

Still, we believe in rewiring for efficient snapshotting. However, it can not show its full potential. If we carefully inspect the description of rewired snapshotting in Section 4.3.2 again, we can observe that rewiring implements a *workaround* of the limitations of the OS. We are forced to manually rewire the virtual memory areas described by the VMAs to create a snapshot — because there is no way to copy a virtual memory area. We have to perform another pass over the source VMAs to set the protection using the system call `mprotect` to read-only — instead of setting it directly when copying the virtual memory area. It is also expensive to keep track of shared physical pages in the presence of multiple snapshots.

Naturally, rewiring hits the limits of the vanilla kernel. Therefore, in the following Section, we will propose a custom system call that tackles these limitations — leading to a much more straight-forward and efficient implementation of virtual snapshotting, which we will finally use in AnKer.

5 SYSTEM CALL VM_SNAPSHOT

In the previous section, we have seen the limitations of the state-of-the-art kernel. Let us now discuss how we can overcome them by introducing our custom system call `vm_snapshot`. In our implementation of rewired snapshotting, we have experienced the need to snapshot virtual memory areas directly. By default, the kernel does not support this task. As a workaround, we had to rewire a new virtual memory area in the same way as the source area which is a costly process as it involves repetitive calls to `mmap`.

5.1 Semantics

To solve this problem, we have to introduce a new system call, that will be the core of our snapshotting mechanism. Before, let us precisely define what *snapshotting a virtual memory area* means in this context. Let us assume we have a mapping from n virtual to n physical pages starting at virtual address b. The first virtual page covering the virtual address space $[b; b+p-1]$ ($vpage_{b0}$) is mapped to the physical page $ppage_{42}$. The second virtual page covering virtual address space $[b+p; b+2\cdot p-1]$ ($vpage_{b1}$) is mapped to another physical page $ppage_7$ and so on.

Now, we want to create a new virtual memory area starting at a virtual address c, that maps to the *same* physical pages. Thus, the virtual page covering $[c; c+p-1]$ should map to $ppage_{42}$, the virtual page $[c+p; c+2\cdot p-1]$ should map to $ppage_7$ and so on. We define the following system call to encapsulate the described semantics:

```
void* vm_snapshot(void* src_addr, size_t length);
```

This system call takes the `src_addr` of the virtual memory area to snapshot and the `length` of the area in bytes. Both `src_addr` and `length` must be page aligned. It returns the address of a new virtual memory area of size `length`, that is a snapshot of the virtual memory area starting at `src_addr`. The new memory area uses the same update semantics as the source memory area, i.e., if the virtual memory area at `src_addr` has been declared using `MAP_PRIVATE | MAP_ANONYMOUS`, the new memory area is declared in the same way. Besides, the new memory area follows the same NUMA allocation policy as any virtual memory of the system – by default, the physical page serving a COW is allocated on the NUMA region of the socket, that executes the thread causing the COW.

5.2 Implementation

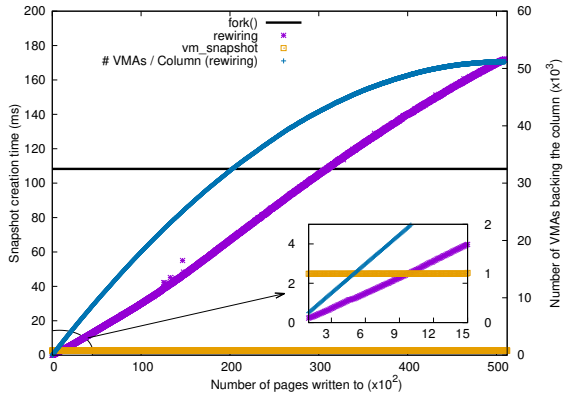
Implementing a system call that modifies the virtual memory subsystem of Linux is a delicate challenge. In the following, we will provide a high-level description of the system call behavior. For the interested reader, we provide a more detailed discussion in Appendix A. On a high level, `vm_snapshot` internally performs the following steps: (1) Identify all VMAs that describe the virtual memory area [`src_addr`, `src_addr + length − 1`]. (2) Reserve a new virtual memory area of size `length` starting at virtual address `dst_addr`. (3) Copy all of the previously identified VMAs and update them to describe the corresponding portions of virtual memory in [`dst_addr`, `dst_addr + length − 1`]. (4) For each VMA which describes a private mapping (which is the standard case in AnKer), additionally copy all existing PTEs and update them to map the corresponding virtual pages in [`dst_addr`, `dst_addr + length − 1`]. This system call `vm_snapshot` will form the core component of creating snapshots on columns in AnKer. It is the call that we use in Figure 1 in Step ④ and Step ⑦.

5.3 Evaluation

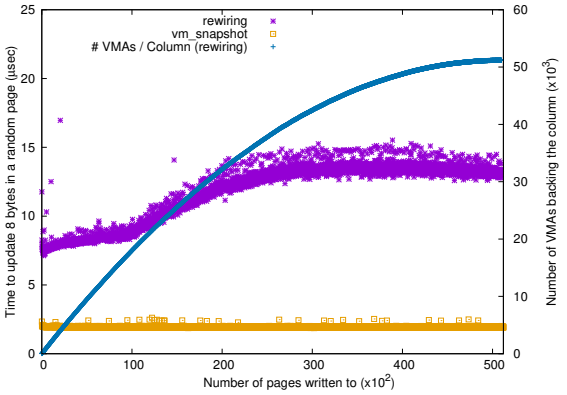
Let us now see how our custom system call `vm_snapshot` performs in comparison with its direct competitor rewiring. We excluded the baseline of physical snapshotting, as it is already out of consideration for AnKer due to high cost and low flexibility. We first look at the snapshot creation time for a single column of 200MB. The previous experiment presented in Table 1 showed that rewiring is profoundly influenced by the number of VMAs that are backing the column to snapshot. To analyze this behavior in comparison with `vm_snapshot`, we run the following experiment: for each of the 51,200 pages of the column, we perform precisely one write to the first 8B of the page. In the case of rewiring, this write triggers the COW of the touched page and thus, creates a separate VMA describing it. After each write, we create a new snapshot of the column and report the creation time.

In Figure 4(a), as predicted, the snapshot creation cost of rewiring is highly influenced by the number of VMAs that is increasing with every modified page. To visualize this correlation, we plot the number of VMAs per column for rewiring alongside with the snapshot creation time. In contrast to rewiring, our system call `vm_snapshot` shows both a stable and low runtime over the entire sequence of

writes. After only around 1000 writes have happened (see zoom-in of Figure 4(a)), the snapshotting cost of `vm_snapshot` already becomes lower than the one of rewiring. After all 51,200 writes have been carried out, `vm_snapshot` is 68x faster than rewiring. This shows the tremendous effect of avoiding repetitive calls to `mmap`.



(a) **Comparison of snapshot creation times**. The time to snapshot a single column is shown on the left y-axis for rewiring respectively `vm_snapshot`. To enhance the visualization, we also show a zoom-in.



(b) **Comparison of writes** to the snapshotted column. On the left y-axis, the time to perform a write of 8B is shown.

Figure 4: Comparison of vm_snapshot and rewiring in terms of snapshotting and write cost. After every write to a page, a new snapshot is taken. Additionally, we show the number of VMAs per column for rewiring on the right y-axis.

However, we should also look at the actual cost of writing the virtual memory. In the case of rewiring, the triggered COW is handled by copying the page content to an unused page and rewiring that page into the column. In the case of `vm_snapshot`, which works on anonymous memory and relies on the COW mechanism of the operating system, no manual handling is necessary. This becomes visible in the runtime shown in Figure 4(b). Writing a page of the column snapshotted by `vm_snapshot` is up to 6x faster than writing to one created by rewiring, as the operating system handles the entire COW. No protection must be set manually, and no signal handler is necessary to detect the write to a page.

6 EXPERIMENTAL EVALUATION

After the description of the processing concept of AnKer and the introduction of `vm_snapshot` to efficiently snapshot virtual memory areas, let us now start with the experimental evaluation of the actual system. As AnKer relies on a hybrid processing model, we want to test it against MVCC using a single execution engine. Additionally, we want to test its snapshotting capabilities against fork-based snapshotting. Our prototype is designed in a way also to support both hybrid processing using `fork` as well as MVCC using a single execution engine by disabling snapshotting.

6.1 System Configurations

Let us define the precise configurations we are going to evaluate:
(1) MVCC in a Single Execution Engine, Full Serializability (abbreviated by SEE_FS). We configure our prototype such that *no snapshots* are taken at all. Thus, there is only a single execution engine with the most recent representation of the database. Both OLTP transactions and OLAP queries run on this execution engine under *full serializability* guarantees. A separate garbage collection mechanism cleans the version chains created by the updates. The system uses a thread that passes over the version chains every second and deletes all versions that are not visible to the oldest active transaction in the system. To speed up scanning over versioned data, we apply an optimization technique introduced by [16]: for every 1024 rows, we keep the position of the first and the last versioned row. With this information, it is possible to scan in tight loops between versioned records without performing any checks.
(2) MVCC in a Single Execution Engine, Snapshot Isolation (abbreviated by SEE_SI). As in (1), *no snapshots* are taken. There is only a single execution engine with the most recent representation of the database. Both OLTP transactions and OLAP queries run in this component under *snapshot isolation* guarantees and thus, no read set validation is performed. The same garbage collection and scan optimization as in (1) are applied.
(3) MVCC in a Single Execution Engine, Read Uncommitted (abbreviated by SEE_RU). As in (1) and (2), *no snapshots* are taken. There is only a single execution engine with the most recent representation of the database. Both OLTP transactions and OLAP queries run in this component under *read uncommitted* guarantees and thus, running transactions/queries can see uncommitted changes. Not garbage collection is necessary since updates do not create versions. Scan optimization is not necessary as well.
(4) MVCC in a Hybrid Execution Engine using `vm_snapshot`, Full Serializability (abbreviated by HEE_AnKer). The OLTP transactions run in the OLTP execution engine, and the OLAP queries run in the OLAP execution engine. The creation of snapshots works in a lazy fashion using our system call `vm_snapshot` as described in Section 3.2.1. We additionally force the transactions, that are classified as OLTP to abort, as soon as they are forced to find the right tuple version from the version chain. This prevents these transactions from doing unnecessary work before they abort.
(5) MVCC in a Hybrid Execution Engine using `fork`, Full Serializability (abbreviated by HEE_fork). Same as (4), except that we use `fork` to perform the virtual snapshotting instead of `vm_snapshot`. A call to `fork` launches a new process of AnKer that runs the OLAP queries. To create a consistent snapshot, HEE_fork blocks all commits until the fork is complete. This can be replaced

with log-based rollback similar to HyPer[10] to improve the OLTP throughput, but it adds additional cost to snapshot preparation.

6.2 Experimental Setup

To evaluate the system under a complex HTAP workload, we define the following mixture of OLTP transactions and OLAP queries:

On the OLTP side, we use three transactions from the TPC-C benchmark: `Payment`, `NewOrder`, and `OrderStat`. These three transactions access all nine tables of the TPC-C database and perform updates on the tables `stock`, `order_line`, `orders`, `new_order`, and `district`. For each transaction that is submitted to the system, we pick the configuration parameters randomly within the bounds given in the TPC-C specification. We populate the database with 40 warehouses. We support two different types of access pattern for the transactions. For the first one, the accesses are *uniformly* distributed across warehouses and districts. For the second one, 50% of the accesses are *skewed* towards five given warehouse/district pairs. The remaining 50% follow a uniform distribution.

On the side of OLAP, we use eight synthetic queries, which are dominated by scanning, grouping, and aggregation. Figure 5 shows the precise queries. We pick the query `StockScan` (OLAP-Q1) as described in [25], which operates on the `warehouse` and `stock` tables. Further, we add the two single-table queries OLAP-Q2 and OLAP-Q3, which group and aggregate on `order_line`. To have scan-heavy queries, we add OLAP-Q4 and OLAP-Q5, which simply perform full table scans on `orders` respectively `new_order`. Finally, we add the three fast queries OLAP-Q6, OLAP-Q7, and OLAP-Q8, which perform scans and aggregations on the columns of a single table.

Unless mentioned otherwise, the upcoming experiments use 6 threads to process the stream of incoming OLTP transactions and 2 threads to answer OLAP queries.

OLAP-Q1 StockScan [25]

```
select w_id, count(*)
  from warehouse, stock
 where w_id = s_w_id
group by w_id;
```

OLAP-Q2

```
select ol_d_id,
       avg(ol_amount)
  from order_line
group by ol_d_id;
```

OLAP_Q3

```
select ol_w_id,
       sum(ol_quantity),
       avg(ol_amount)
  from order_line
group by ol_w_id;
```

OLAP-Q4 FULL TABLE SCAN

```
select *
  from orders;
```

OLAP-Q5 FULL TABLE SCAN

```
select *
  from new_order;
```

OLAP Q6 COLUMN SCAN

```
select avg(d_tax),
       avg(d_ytd)
  from district;
```

OLAP Q7 COLUMN SCAN

```
select avg(ol_amount)
  from order_line;
```

OLAP Q8 COLUMN SCAN

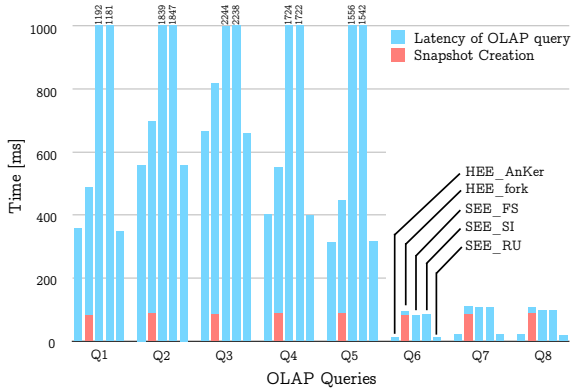
```
select avg(s_quantity)
  from stock;
```

Figure 5: The eight OLAP queries we use in the evaluation.

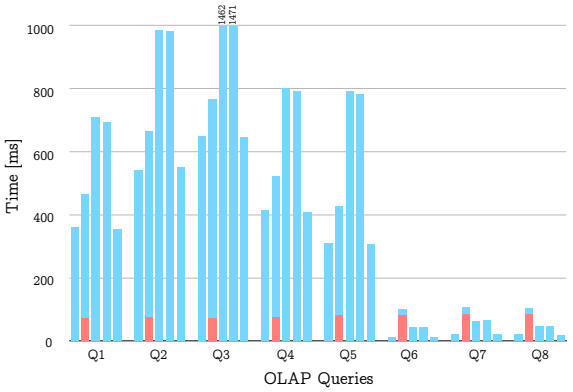
6.3 Snapshotting Cost and OLAP Latency

Let us start with an evaluation of the core mechanism of AnKer: the fast snapshotting using our system call `vm_snapshot`. Our initial motivation of this project was to enable virtual snapshotting without the overhead of fork-based snapshotting. Thus, let us now first see how hybrid processing using our system call (HEE_AnKer) competes with hybrid processing using `fork` as originally done by HyPer (HEE_fork). Both run under full serializability guarantees. Additionally, we compare the two hybrid approaches with MVCC using a single execution engine under three different isolation levels (SEE_FS, SEE_SI, SEE_RU).

In the following experiment, we will answer two questions: First, how expensive is the snapshotting mechanism using our system call in comparison with the alternatives under a real-world HTAP workload? Second, what is the impact of the snapshotting mechanism and the hybrid processing under MVCC on the latency of the OLAP queries? To answer these questions, we perform the following experiment: To sustain the system, we fire an infinite stream of OLTP transactions randomly picked from the set of TPC-C transactions. After five seconds, we fire a random OLAP query from the set depicted in Figure 5 and repeat firing random OLAP queries every 500ms. For every fired query, we create a new snapshot for the hybrid approaches HEE_AnKer and HEE_fork. After three minutes, we terminate the experiment and report the average of all observed snapshotting times and query latencies.



(a) System sustained using OLTP transactions with **uniform access pattern**.



(b) System sustained using OLTP transactions with **skewed access pattern**.

Figure 6: Snapshotting Cost and Latency of OLAP queries.

Figure 6 shows the results grouped by the OLAP queries. In Figure 6(a), we use the OLTP workload with a uniform access pattern while in Figure 6(b), we use the skewed OLTP workload focusing on five hot warehouse/district combinations. For each of the eight OLAP queries, we report the snapshot creation time as well as the latency in ms for each of the five tested methods.

Let us first have a look at the results on the uniform pattern in Figure 6(a). If we compare the baselines, we can see the significant cost of the snapshotting phase in HEE_fork, caused by replicating the process, followed by a fast query answering part. The approaches SEE_FS and SEE_SI using a single execution engine do not have an explicit snapshot creation phase but suffer from very high query execution times of up to 2200ms for Q_3 (cut off at 1000ms in the plot), as the OLAP query has to work its way through the version chains, which are build up by the OLTP stream. In comparison to the baselines, HEE_AnKer combines the best of

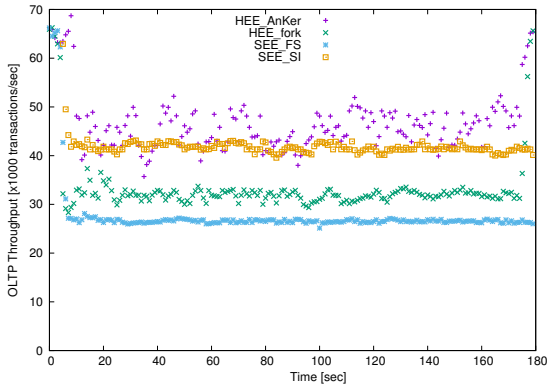
both worlds: its snapshot creation time is so short that it is not even visible in the plots. This is caused by the fact that it snapshots only the columns that are touched by the respective query using `vm_snapshot`. In comparison to HEE_fork, the snapshotting phase of HEE_AnKer is more than 100x faster. After snapshotting, the actual query answering part is equally fast as for HEE_fork. We want to point out that the total latency for HEE_AnKer (including snapshot creation and query answering) under full serializability guarantees almost equals the runtime of SEE_RU, which runs only on the isolation level of read-uncommitted. We can also observe that depending on the query, the expensive snapshotting phase of HEE_fork can have a drastic impact on the overall latency. For Q_6, Q_7, and Q_8, the snapshotting cost of HEE_fork dominates the latency, and HEE_AnKer achieves a speedup of 4.9x to 7.5x.

If we look at the results on the skewed pattern in Figure 6(b), we can observe that especially the approaches using a single execution engine and a higher isolation level (SEE_FS and SEE_SI) massively benefit from the skew. We can also see that for the three faster queries Q_6, Q_7, and Q_8, HEE_fork shows now the overall highest latency: the snapshotting mechanism using `fork` is more expensive than the query answering part of any competitor.

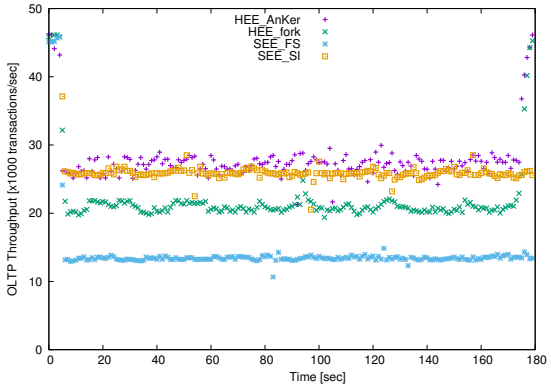
6.4 Transaction Throughput

After inspecting the snapshotting cost and the latency of the OLAP queries, let us now investigate the previous experiment from Section 6.3 from the OLTP side. As we fire an infinite stream of OLTP transactions over a time interval of three minutes, we plot the OLTP throughput achieved per second. As before, starting after five seconds, we fire a random OLAP query every 500ms and snapshot before every OLAP query for the hybrid methods. We exclude the results for SEE_RU, as the OLTP throughput is extremely high in comparison to the counterparts with higher isolation levels.

Figure 7 shows the results for the uniform OLTP access pattern (Figure 7(a)) and the skewed OLTP access pattern (Figure 7(b)). For the uniform case, in the first five seconds, no OLAP query is running, and we see the maximum OLTP throughput of the system which locates between 60k and 70k transactions per second. As soon as the first OLAP query arrives, the throughput significantly drops for all methods due to congestion. As expected, we observe the lowest throughput of around $28k$ transactions per second under SSE_FS because of the expensive commit phase validation that is performed, and a slightly higher throughput of around 30k to 38k transactions per second for HEE_fork. The costly snapshotting utilizing `fork` heavily throttles the OLTP throughput. The throughput of SEE_SI is very stable around 42k transactions per second. To our surprise, this is a lower throughput than the average throughput achieved by HEE_AnKer with 47k transactions per second which has one major reason: Before SEE_SI successfully reads a value that is stored in a column, it must validate whether the value is versioned or not. If it is versioned, it reads the timestamp of the most recent version and the pointer to the version chain. The read is successful if the current timestamp of the tuple is smaller than transaction's `begin-timestamp`. If not, the version chain is traversed to find the valid version. However, for the HEE_AnKer this is not the case. Since we only support full-serializability, the OLTP transactions are not allowed to read from the version chains. For every read, the transaction only reads the timestamp of the current tuple version.



(a) Throughput of OLTP Transactions with **uniform access pattern**.



(b) Throughput of OLTP Transactions with **skewed access pattern**.

Figure 7: Throughput of OLTP Transactions.

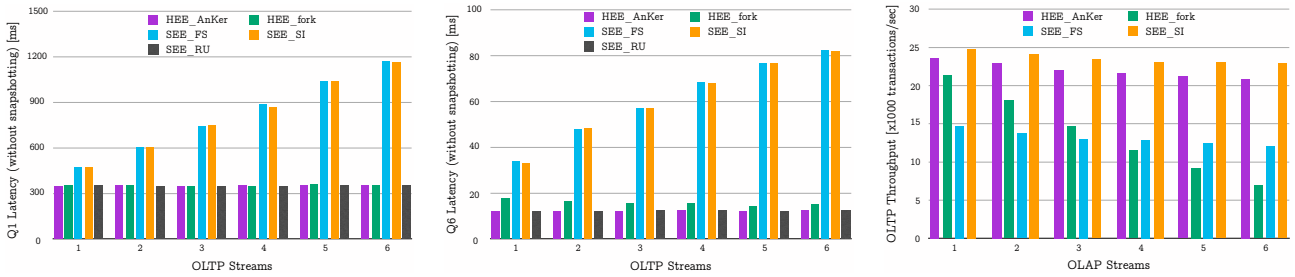
If the timestamp is smaller than transaction's `begin-timestamp`, the transaction can read the value. Otherwise, it prematurely aborts without doing the read. Due to fewer comparisons and metadata validation, a single read access for full-serializable OLTP transaction is faster than the reads performed by SI based protocol where the transaction can still commit after reading from the version chain. For HEE_AnKer, we observe a high variance in the throughput which is due to different snapshot creation time for different OLAP queries and the copy-on-write cost. We also observe a stable behaviour for HEE_Fork due to relatively stable `fork` cost.

For the skewed distribution in Figure 7(b), we see a lower throughput than for the uniform pattern across all methods. This is caused by update conflicts that must be serialized for the contended transactions. We can also observe in this plot that the variance for HEE_AnKer and HEE_fork is smaller than in the uniform case since fewer copy-on-writes are performed.

6.5 Scaling

Our system essentially implements parallelism on two layers: On the first layer, we parallelize OLTP and OLAP execution by maintaining a hybrid execution engine. On the second layer, we apply MVCC inside each engine to ensure a high concurrency among transactions of a single type. In this regard, let us now investigate how well AnKer scales with the number of OLTP and OLAP streams, that are used to process the transactions and queries.

In Figure 8, we investigate the scaling capabilities along two dimensions. On the first dimension, we fix the number of OLAP



(a) **Latency for Q1** while **varying the number of OLTP streams**. The number of OLAP streams is 2.

(b) **Latency for Q6** while **varying the number of OLTP streams**. The number of OLAP streams is 2.

(c) **OLTP Throughput** while **varying the number of OLAP streams**. The number of OLTP streams is 2.

Figure 8: Varying the number of streams used for processing.

streams and vary the number of available OLTP streams. On the second dimension, we fix the number of OLTP streams and vary the number of available OLAP streams. The experimental setup is the same as in Section 6.3 and Section 6.4. In Figure 8(a) and Figure 8(b), we report the latency of the OLAP queries Q_1 and Q_6 when fixing the number of OLAP streams to 2 and varying the number of OLTP streams from 1 to 6 to sustain the system with varying OLTP load. In Figure 8(c), we fix the OLTP streams to 2 and vary the number of OLAP streams from 1 to 6. Here, we show the average throughput over the run of 180 seconds.

From Figure 8(a) and Figure 8(b), we can see that the hybrid approaches are largely unaffected by the number of OLTP streams. The reason for this is that the OLAP processing happens in isolation to the version chain building, that is happening in the OLTP execution engine. In contrast to that, the OLAP latency of SEE_FS and SEE_SI heavily decreases with an increase in OLTP streams, as more OLTP streams build up more version chains that must be traversed by an OLAP query. SEE_RU is again unaffected, as it simply reads the in-place version without traversing the version chains at all. In Figure 8(c), we can see that the OLTP throughput decreases for all methods with an increase of OLAP streams. However, some methods are more affected than others. While the throughput of HEE_AnKer decreases only by 11.6% from 1 to 6 OLAP streams, the throughput of HEE_fork decreases by 67.6%. This is because the expensive snapshotting phase using `fork` interrupts the processing of the OLTP stream for a significant amount of time and decreases the number of OLTP transactions that can be processed in 180 seconds. Consequently, the effective throughput is decreased.

Table 2: Varying the number of warehouses and observing the throughput decrease. The throughput is given in transactions per second. The last column shows the slowdown in throughput from 1 warehouse to 40 warehouses.

Method	1 WH	10 WH	20 WH	30 WH	40 WH	Slowdown 1WH→40WH
HEE_AnKer	51289	50525	49718	48637	47729	1.07x
HEE_fork	46391	42741	39172	35678	31220	1.49x
SEE_FS	32456	31810	31281	30299	28794	1.18x
SEE_SI	48391	48027	47687	47033	46237	1.05x

Finally, let us vary the size of the used dataset and see the effect on the OLTP throughput. Additionally to 40 warehouses, that we used in the previous experiments, we also evaluate 1, 10, 20, and 30 warehouses in the following and report the slowdown in throughput when increasing the size from 1 to 40 warehouses. As expected, HEE_fork is affected the most by an increase of the dataset size with a throughput slowdown of factor 1.49x, as the process to fork heavily increases in size. For HEE_AnKer, the problem is not that severe as only the touched columns are snapshotted.

7 FUTURE WORK

Our system call `vm_snapshot` is the essential component that powers the hybrid execution engine of AnKer. It enables fast and fine-granular snapshotting in combination with a low memory footprint. Nevertheless, due to its flexibility and general design, it could be applied in a variety of other situations as well. From a more general perspective, `vm_snapshot` can essentially replace any larger **memcpy** operation. While `memcpy` duplicates *all* pages in a memory region in an eager fashion, `vm_snapshot` lazily duplicates only the *modified* pages. As `memcpy` is frequently used at essentially all levels of any software system, such a simple function swap can have a significant impact on performance. From a system perspective, the problem of efficient snapshotting is not limited to relational systems. For instance, **graph processing systems** throttle in the presence of concurrent updates and analytics. Our system call could be used to snapshot (parts of) the graph and outsource the analytics as in AnKer. Apart from snapshotting, there is the related concept of **checkpointing** [24], where a consistent view of the database has to be stored to disk for recovery purposes. As this is more of a background task with respect to the query processing, it should be as transparent and lightweight as possible. We can ensure this with our system call: if a checkpoint is requested, we create a consistent view of the database using `vm_snapshot` with minimal effort. Afterwards, this view can be spilled to disk asynchronously. Consequently, a system, which supports the creation of snapshots and checkpoints at a high frequency, could be easily extended to run **time travel queries** efficiently: they would either run on snapshots, which are still available in the system, or on checkpoints, that are reloaded via `mmap`.

8 CONCLUSION

In this work, we introduced AnKer, a transactional processing concept implementing a hybrid execution engine in combination with MVCC, which works hand in hand with our customized Linux kernel to enable snapshotting at a very high frequency. We have shown that a hybrid design powered by a lightweight snapshotting mechanism fits naturally to the HTAP workloads and improves the throughput of OLAP queries by factors up to 4x, as it enables fast scans in tight loops. Besides, due to the flexibility of our custom system call `vm_snapshot`, we can limit the snapshotting effort to those columns that are accessed by transactions, allowing a snapshotting speedup of more than factor 100x over fork-based snapshotting.

REFERENCES

[1] 2017. MemSQL. (10 2017). http://www.memsql.com
[2] 2017. MySQL. (10 2017). http://www.mysql.com
[3] 2017. NuoDB: http://www.nuodb.com. (10 2017). http://www.nuodb.com
[4] 2017. Peloton: http://www.pelotondb.org. (10 2017). http://www.pelotondb.org
[5] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. 1987. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley. http://research.microsoft.com/en-us/people/philbe/ccontrol.aspx
[6] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Åke Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. 2013. Hekaton: SQL server's memory-optimized OLTP engine. In *SIGMOD 2013, New York, NY, USA, June 22-27, 2013.* 1243–1254. https://doi.org/10.1145/2463676.2463710
[7] Franz Färber, Sang Kyun Cha, Jürgen Primsch, Christof Bornhövd, Stefan Sigg, and Wolfgang Lehner. 2011. SAP HANA database: data management for modern business applications. *SIGMOD Record* 40, 4 (2011), 45–51. https://doi.org/10.1145/2094114.2094126
[8] Alan Fekete, Elizabeth J. O'Neil, and Patrick E. O'Neil. 2004. A Read-Only Transaction Anomaly Under Snapshot Isolation. *SIGMOD Record* 33, 3 (2004), 12–14. https://doi.org/10.1145/1031570.1031573
[9] Jana Giceva, Gerd Zellweger, Gustavo Alonso, and Timothy Rosco. 2016. Customized OS Support for Data-processing. In *DaMon' 16.* ACM, New York, NY, USA, Article 2, 6 pages. https://doi.org/10.1145/2933349.2933351
[10] A. Kemper and T. Neumann. 2011. HyPer: A hybrid OLTP & OLAP main memory database system based on virtual memory snapshots. In *ICDE 2011.* 195–206. https://doi.org/10.1109/ICDE.2011.5767867
[11] Kangnyeon Kim, Tianzheng Wang, Ryan Johnson, and Ippokratis Pandis. 2016. ERMIA: Fast Memory-Optimized Database System for Heterogeneous Workloads. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016.* 1675–1687. https://doi.org/10.1145/2882903.2882905
[12] Per-Åke Larson, Spyros Blanas, Cristian Diaconu, Craig Freedman, Jignesh M. Patel, and Mike Zwilling. 2011. High-Performance Concurrency Control Mechanisms for Main-Memory Databases. *PVLDB* 5, 4 (2011), 298–309.
[13] Qingzhong Meng, Xuan Zhou, Shiping Chen, and Shan Wang. 2016. SwingDB: An Embedded In-memory DBMS Enabling Instant Snapshot Sharing. In *ADMS/IMDM Workshop 2016.* 134–149. https://doi.org/10.1007/978-3-319-56111-0_8
[14] C. Mohan, Hamid Pirahesh, and Raymond A. Lorie. 1992. Efficient and Flexible Methods for Transient Versioning of Records to Avoid Locking by Read-Only Transactions. In *SIGMOD 1992.* 124–133. https://doi.org/10.1145/130283.130306
[15] Henrik Mühe, Alfons Kemper, and Thomas Neumann. 2011. How to efficiently snapshot transactional data: hardware or software controlled?. In *DaMoN 2011, Athens, Greece.* 17–26. https://doi.org/10.1145/1995441.1995444
[16] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. 2015. Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems. In *SIGMOD 2015.* 677–689. https://doi.org/10.1145/2723372.2749436
[17] Muhsen Owaida, David Sidler, Kaan Kara, and Gustavo Alonso. 2017. Centaur: A Framework for Hybrid CPU-FPGA Databases. In *FCCM 2017, Napa, CA, USA, April 30 - May 2, 2017.* 211–218. https://doi.org/10.1109/FCCM.2017.37
[18] Dan R. K. Ports and Kevin Grittner. 2012. Serializable Snapshot Isolation in PostgreSQL. *PVLDB* 5, 12 (2012), 1850–1861.
[19] Felix Martin Schuhknecht, Jens Dittrich, and Ankur Sharma. 2016. RUMA has it: Rewired User-space Memory Access is Possible! *PVLDB* 9, 10 (2016), 768–779.
[20] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy transactions in multicore in-memory databases. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013.* 18–32. https://doi.org/10.1145/2517349.2522713
[21] Annett Ungethüm, Dirk Habich, Tomas Karnagel, Sebastian Haas, Eric Mier, Gerhard Fettweis, and Wolfgang Lehner. 2017. Overview on Hardware Optimizations for Database Engines. In *BTW 2017, 6.-10. März 2017, Stuttgart, Germany, Proceedings.* 383–402.
[22] Tianzheng Wang, Ryan Johnson, Alan Fekete, and Ippokratis Pandis. 2017. Efficiently making (almost) any concurrency control mechanism serializable. *The VLDB Journal* 26, 4 (01 Aug 2017), 537–562. https://doi.org/10.1007/s00778-017-0463-8
[23] Tianzheng Wang and Hideaki Kimura. 2016. Mostly-Optimistic Concurrency Control for Highly Contended Dynamic Workloads on a Thousand Cores. *PVLDB* 10, 2 (2016), 49–60. http://www.vldb.org/pvldb/vol10/p49-wang.pdf
[24] Gerhard Weikum and Gottfried Vossen. 2002. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery.* Morgan Kaufmann.
[25] Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. 2017. An Empirical Evaluation of In-Memory Multi-Version Concurrency Control. *PVLDB* 10, 7 (2017), 781–792.
[26] Xiangyao Yu, George Bezerra, Andrew Pavlo, Srinivas Devadas, and Michael Stonebraker. 2014. Staring into the Abyss: An Evaluation of Concurrency Control with One Thousand Cores. *PVLDB* 8, 3 (2014), 209–220.

A VM_SNAPSHOT IMPLEMENTATION DETAILS

For the interested reader, the follow section provides a detailed description of the implementation details of `vm_snapshot`.

```
void* vm_snapshot(void* src_addr,
                  size_t length);
```

(1) Check if the virtual memory area to snapshot in the range [`src_addr`, `src_addr` + `length` − 1] is actually allocated. If no, the call fails with return value `MAP_FAILED` and sets `errno` accordingly.
(2) Allocate a fresh virtual memory area [`dst_addr`, `dst_addr` + `length` − 1] that will map to the pages backed by the source virtual memory area.
(3) Identify all VMAs that describe the virtual memory area [`src_addr`, `src_addr` + `length` − 1]. This might be one VMA or multiple ones. Let us call them in the following VMA_0 to VMA_{n-1}, if n VMAs describe the area.
(4) It is possible that VMA_0 and VMA_{n-1}, the VMAs describing the borders of the virtual memory area, span larger than the area to replicate. This can be the case if virtual memory before `src_addr` or after `src_addr`+`length` is currently allocated as well. In this case, we split VMA_0 and VMA_{n-1} at `src_addr` respectively `src_addr` + `length`. If a split happens, we update VMA_0 and VMA_{n-1} to the VMAs that now exactly match the borders of the region to replicate.
(5) Iterate over VMA_0 to VMA_{n-1}. Let us refer to the current item as VMA_i. Further, let us define `size`(VMA_i) as the size of the described virtual memory area and `offset`(VMA_i) as the address of the described virtual memory area relative to `src_addr`. Now, we create an exact copy of VMA_i and update the virtual memory area described by it to [`dst_addr` + `offset`(VMA_i), `dst_addr` + `offset`(VMA_i) + `size`(VMA_i)).
(6) Further, we check whether VMA_i describes a shared or a private virtual memory area. If VMA_i is shared, nothing more has to be done for this VMA. If VMA_i is private, we additionally have to modify the page table, if there exist PTEs for the virtual memory area that VMA_i is describing. In this case, we identify all k PTEs, which relate to VMA_i, as PTE_0 to PTE_{k-1}.
(7) Iterate over PTE_0 to PTE_{k-1}. Let us refer to the current item as PTE_j. If `pageoffset`(PTE_j) returns the address of the mapped virtual page relative to `src_addr`, we create a copy of PTE_j and update the start address of the mapped virtual page in the copy to `dst_addr` + `pageoffset`(PTE_j). This step is necessary for private VMAs, as any write that is happening to the described virtual memory area results in a copy-on-write, that is handled with an anonymous physical page. As the information about the physical page is not present in the VMA but only in the corresponding PTE, we have to modify the page table in this case.
(8) In the end, we update the statistics kept by the kernel that tracks the VM area reserved by the process.

After these steps, the virtual memory area [`dst_addr`, `dst_addr` + `length` − 1] contains the snapshot and can be accessed.