No Time to Halt: In-Situ Analysis for Large-Scale Data Processing via Virtual Snapshotting

Reza Salkhordeh
rsalkhor@uni-mainz.de
Johannes Gutenberg University
Mainz, Germany

Felix Schuhknecht
schuhknecht@uni-mainz.de
Johannes Gutenberg University
Mainz, Germany

Hossein Asadi
asadi@sharif.edu
Sharif University of Technology
Tehran, Iran

Steffen Eiden
seiden@students.uni-mainz.de
Johannes Gutenberg University
Mainz, Germany

André Brinkmann
brinkman@uni-mainz.de
Johannes Gutenberg University
Mainz, Germany

ABSTRACT

Large-scale data processing applications often perform long running computations or analytical queries over days or even weeks. While these are running, users typically get no proper information on the current state and progress of the computation and the quality of the intermediate result. Consequently, it takes a long time until a user is able to assess whether a computation was useful or not. To shorten this feedback loop, *in-situ analysis* accesses the internal state of the application *while* it is still running. Unfortunately, state-of-the-art techniques come with two problems: They (a) require the application to halt in order to access a stable state and (b) require extensive code modification.

To solve both problems, we advocate to perform virtual snapshotting instead, which allows to create stable snapshots without halting or modifying the application in any way using a copy-on-write based approach. While we already show-cased the core technique as a proof-of-concept in previous work, it remains open how effectively our technique coined in-situ-CoW actually operates on different types of applications. To find out, we first perform an in-depth analysis of memory access patterns of 16 large-scale data processing workloads, including representative applications from the scientific domain as well as from data management. Revealing highly different needs of application classes, we adjust in-situ-CoW to dynamically adapt the snapshotting granularity to the workload. In an extensive experimental evaluation, we compare in-situ-CoW on all applications against (i) traditional physical snapshotting, (ii) MVCC, as well as (iii) two oracle policies. In comparison to traditional physical snapshotting, in-situ-CoW reduces the performance overhead by up to 98% (66% on average) for scientific workloads and by up to 64% (45% on average) for YCSB. In comparison to MVCC, in-situ-CoW reduces the performance overhead by up to 89% (25% on average) under write-intensive workloads.

1 INTRODUCTION

Large-scale data processing in the area of big data faces enormous workloads to execute [22, 24]. For example, workloads from the scientific context [2, 3, 5, 15, 17–19, 25, 26, 34], such as physical simulations, process large volumes of data until a final result is eventually produced. As these computation can easily take hours, days, or even weeks, it takes equally long until the user is able to inspect the result, assess its quality, and potentially restart the execution with a different parameter set.

To shorten this feedback loop, in-situ processing [7, 36] was proposed. The core idea is creating a stable snapshot of the intermediate state of the data processing application while it is still running and expose it. This allows to perform various analysis tasks on these snapshots, such as progress monitoring, early error detection, and the extraction of intermediate results as an approximation of the final one. Further, apart from analysis, getting access to the internal state allows to manually create checkpoints [11, 23] of the state and to persist it to stable storage, which allows recovering in case of a crash[1].

While in-situ processing has the aforementioned advantages, in its traditional form, it unfortunately has two downsides: (a) To create a stable snapshot, the running computation must be halted until the snapshot creation has finished. As for large datasets, the creation of a snapshot can take a considerable amount of time, this slows down the overall data processing significantly. If snapshots are created frequently to have an up-to-date view on the processing, this penalty becomes even more severe. (b) To enable in-situ processing in the first place, the large-scale data processing application must be deeply modified to be able to create snapshots, i.e., by exposing its state to an in-situ framework. This modification can be complex, especially for more sophisticated snapshotting approaches, which snapshot only those portions of the data that have been modified since the last snapshot. Further, there is the possibility that the source code is simply not accessible, rendering the integration of in-situ processing impossible.

1.1 In-situ processing without halting: in-situ-CoW

To address both (a) and (b), we advocate to combine virtual snapshotting with in-situ processing. It consists of two parts: First, instead of creating physical snapshots, we create only *virtual snapshots* which essentially resemble views in virtual memory on the original state. To ensure that these virtual snapshots remain stable under modifications of the state (which can happen at any time since we do not halt the application), we utilize a *copy-on-write* (CoW) mechanism [32] based on memory rewiring [28] which lazily copies only those memory pages that are actually modified by the application. In contrast to other CoW-based mechanisms [20, 30], this approach does not require the operating system (OS) to be changed in any way, nor does it require root privileges. Second, to avoid modifying the code of the data

[1] Note that our method creates snapshots in main-memory. However, this snapshot can be written asynchronously to stable storage to make it usable for crash recovery.

© 2025 Copyright held by the owner/author(s). Published in Proceedings of the 28th International Conference on Extending Database Technology (EDBT), 25th March-28th March, 2025, ISBN 978-3-89318-098-1 on OpenProceedings.org.
Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0.

processing application, we utilize so-called *function interposing*. The core principle is to pre-load a library when the binary of the application is started which then automatically intercepts all memory allocations the application is performing. This gives us a handle on that memory, such that we can create snapshots on it in a minimally invasive fashion without having to modify the source code of the application in any way.

While we showcased the core technique already in a previous work [29], we demonstrated the concept only for a very simplified HTAP workload. However, it remains an open question how well the concept performs for different classes of actual scientific applications, as they have complex memory access patterns, varying access frequencies, and heterogeneous access types. As we will see by the study of eleven different representative data-intensive applications, including the YCSB-benchmark, the pattern severely impacts the *granularity* at which CoW should be performed as well as the frequency at which snapshots should be created, aka the *interval time*. Consequently, we propose an adaptive adjustment of the CoW granularity depending on the observed access pattern. Further, to reduce the impact of performing the CoW of a memory page, we propose to copy the page *asynchronously* to the continuous execution of the application. Only if the applications needs to modify a page again while it is still being copied, it has to halt for moment.

1.2 Contributions and Structure of the Paper

In summary, we make the following **contributions**:

(1) To evaluate whether our CoW-based approach would operate effectively in practice, we first monitor and analyze ten representative scientific applications [3, 15, 17, 25, 26] as well as six workloads of the YSCB benchmark under Redis in terms of their **memory access patterns**. We pose and answer the following research questions: (i) Can a CoW-based approach have a lower memory overhead than physical snapshotting? (ii) Is it sufficient to perform CoW on the system page size or should other granularities be considered? (iii) How should the snapshotting time interval be set? (iv) Is there a trade-off between memory overhead and the processing overhead when setting the CoW granularity and snapshotting time interval? (v) Can we categorize applications with respect to their access pattern? Our analysis reveals that the optimal CoW configuration is highly heterogeneous across applications.

(2) Based on our findings, we extend our original prototype of **in-situ virtual snapshotting** from [28, 29] in three ways: (i) We support arbitrary CoW granularities, as long as they are a multiple of the system page size. (ii) We allow to adaptively adjust the granularity at which CoW is performed to fit to the specific memory access pattern of the workload. By this, it aims at keeping both runtime and memory overhead as low as possible. To the best of our knowledge, this is the *first* adaptive snapshotting scheme proposed for large-scale applications. (iii) We perform the copying asynchronously to the continuous execution of the application. Only if a conflict occurs, the application has to wait for a moment.

(3) We perform an **extensive experimental evaluation** of the resulting method in-situ-CoW on all 16 workloads against a set of competitive baselines. We compare against (i) physical snapshotting, copying the entire state under halting, (ii) MVCC-based snapshotting and (iii) against two oracle implementations, which represent the minimum possible performance and memory overhead. We show that in-situ-CoW can significantly reduce the snapshotting overhead over the baselines and the demonstrate the effectiveness of our extensions.

Table 1: Workload characteristics (NoMA: Number of Memory Accesses, WSS: Working Set Size)

Workload	NoMA	WSS [GB]	Application Domain
amg [17]	143M	21.27	Numerical analysis
miniGhost [3]	41M	19.39	Stencil computation
hpl [25]	139M	3.68	Linpack benchmark
lulesh [15]	63M	39.32	Hydrodynamics
milc [5]	62M	2.01	Lattice computation
namd [26]	27M	1.02	Molecular dynamics
nas_ft [2]	102M	77.65	Discrete 3D FFT
snap [18]	173M	25.17	Radiometry
tealeaf [19]	154M	0.68	Thermal conduction
xs_bench [34]	162M	9.46	Neutron transport
YCSB Load	38M	1.3	Database Management
YCSB Workload A	21M	1.3	Database Management
YCSB Workload B	21M	1.3	Database Management
YCSB Workload D	23M	1.3	Database Management
YCSB Workload E	95M	1.3	Database Management
YCSB Workload F	3M	1.3	Database Management

The paper is structured as follows: In Section 2, we first perform a characterization for our 16 data-intensive workloads of interest. Based on these findings, in Section 3, we present in-situ-CoW and all integrated extensions. In Section 4, we perform an extensive experimental evaluation for a large variety of configurations and baselines. In Section 5, we discuss how a potential analysis workflow could look like. In Section 6, we discuss the related work, before concluding in Section 7.

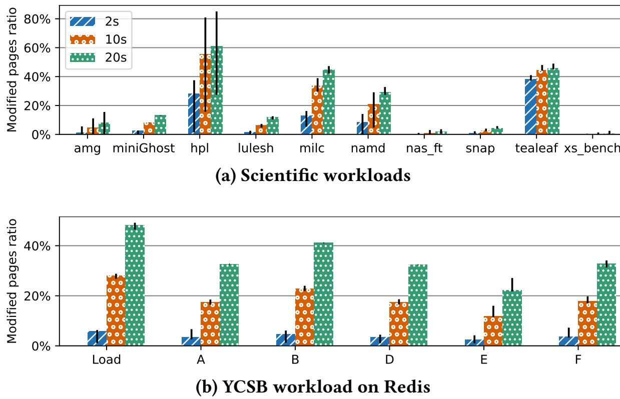
2 WORKLOAD CHARACTERIZATION

Whether CoW-based snapshotting can operate effectively in terms of performance and memory overhead highly depends on the memory access pattern of the application. We therefore first measure the number of pages modified in different time intervals. We then bring our observations into the bigger picture and categorize our 16 workloads into *low overhead*, *high overhead*, and *time interval dependent* under CoW.

2.1 Experimental Setup and Workloads

For all experiments, we use a server with a single 10-core 2.50 GHz Xeon Gold processor with 192 GB of main memory. We dedicate eight cores to the execution of the application and reserve the other two cores to capture the memory accesses. We use *perf* [10] to capture memory access traces with a 100ns granularity.

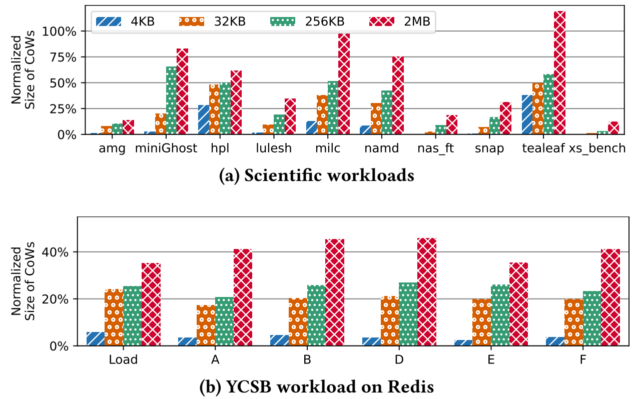
We selected ten large-scale data intensive applications from the domain of scientific computing, where we included an application from each of the "seven dwarfs of HPC" [1]. Table 1 provides an overview over the examined workloads. *amg* [17] solves parallel algebraic multigrid for linear systems. *miniGhost* [3] mimics a 3D nearest neighbor communications. *hpl* [25] solves dense linear systems in double precision. *lulesh* [15] simulates hydrodynamics equations by partitioning the spatial domain. *milc* [5] enables simulation of 4D lattice gauge theory. *namd* [26] simulates biomolecular systems. *nas_ft* [2] is a 3D fourier transformer, which uses all-to-all communication. *snap* [18] is a simulator for the performance of discrete ordinates neutral particle transport. *tealeaf* [19] is a mini application for iterative sparse linear solvers. *xs_bench* [34] mimics the Monte Carlo particle transport simulations. All applications manage a large state in main memory on which we want to enable an in-situ analysis.



(a) Scientific workloads

(b) YCSB workload on Redis

Figure 1: Modified pages ratio normalized to the working set size for three different time intervals



(a) Scientific workloads

(b) YCSB workload on Redis

Figure 2: Size of copied data for four CoW granularities, normalized to the working set size

Additionally, from the data management domain, we evaluate all workloads of the YCSB benchmark [8] (except workload C which is read-only) on the in-memory key-value store Redis for 10M entries and 1M operations per workload, which we enhanced with in-situ-CoW. Note that interestingly, Redis also relies on virtual snapshotting, however, for creating recovery checkpoints. In contrast to our method, Redis uses the system call `fork()` to create virtual snapshots in form of spawned child processes. Unfortunately, it suffers from several downsides: It requires expensive process spawning, all virtual memory of the process is always snapshotted, processes must be carefully coordinated, and it supports only page size CoW granularity. Unfortunately, we could not include an experimental comparison with fork(), as the virtual snapshotting is deeply buried in the checkpointing mechanism, which materialized the snapshotted state on disk. However, we show that in-situ-CoW is to superior to fork()-based snapshotting, as it allows using a flexible CoW granularity.

2.2 Number of Modified Pages per Time Interval

Using CoW, for each 4KB page that is modified, a copy of the original page is created. Therefore, the memory footprint of an application with a single snapshot is the sum of all state memory plus the memory of all modified, and hence, copied pages. Of course, apart from the memory access pattern, the time interval at which snapshots are taken also affects the memory footprint. Using a longer interval potentially results in more modified pages, and hence, a higher memory footprint. Therefore, in the following, we investigate the number of modified pages for different time intervals to understand for which applications our CoW based approach would have a lower memory overhead than physically copying the entire dataset.

Fig. 1 shows the ratio of modified pages normalized over the working set size (the total number of memory pages accessed during the application runtime). We vary the time interval between snapshot creations and report the average ratio over intervals. We also show the minimum and maximum values observed over all intervals in form of error bars. We see that for six out of ten scientific workloads, the memory overhead is less than 10%, independent of the time interval. This makes these applications very suitable candidates for our approach. Applications such as *milc*, *namd*, and the YCSB workloads are also suitable, if a short interval of 2s is used. In this case, these workloads induce less than 20% of memory overhead. We can also see that the workloads *hpl* and *tealeaf* require at least 40% additional memory even for the shortest interval, as they modify a large amount of pages in a short period of time.

Findings (1). *In terms of expected memory overhead of CoW, we can categorize our applications into three groups: 1) low overhead (< 10% for any interval): amg, miniGhost, lulesh, nas_ft, xs_bench, 2) high overhead (> 40% for any interval): hpl, tealeaf, and 3) time-interval dependent: milc, namd, and YCSB. This shows that in terms of memory overhead, a CoW-based would indeed be beneficial for a subset of the tested applications.*

2.3 Impact of the CoW Granularity

In the previous section, we measured the number of 4KB pages copied. However, there is the possibility to perform CoW using larger granularities as well. For example, if the granularity is set to 256KB, writing a single byte triggers the copying of the surrounding 256KB chunk. In terms of memory overhead, choosing a small granularity (such as 4KB, the size of a memory page) is better. However, a small granularity causes in many CoW operations, resulting in processing overhead. On the other hand, a large granularity reduces the number of CoW operations, but might result in a higher memory overhead if only a portion of the chunk is modified. In the following experiment, we therefore vary the granularity and observe its impact on the memory overhead.

Fig. 2 shows the amount of copied data for four different granularities normalized to the working set size. Values higher than 100% can occur since there might exist 2MB virtual address blocks for which only a small part of them is accessed by the application. For such address blocks, the parts not accessed by the application are not counted towards the working set size, but are still copied by the CoW approach. In workloads like *miniGhost* and *YCSB-A*, increasing the granularity significantly increases the memory overhead. On the other hand, in workloads like *amg*, *xs_bench* and *YCSB-E*, larger granularities still have a low memory overhead.

Findings (2). *The expected memory overhead is highly affected by the CoW granularity, and hence, different applications will require different granularity configurations.*

2.4 Number of Performed CoW Operations

After focusing on the memory footprint of CoW operations, let us now look at the processing overhead, which depends on the number of CoW operations. Inverse to the number of modified pages, this number decreases for longer time intervals, since a page modified multiple times within the same interval will be copied only once.

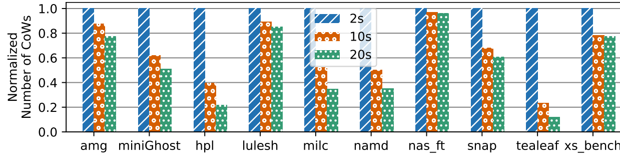


Figure 3: Normalized number of CoWs (4KB granularity for 2/10/20s)

Fig. 3 therefore shows the number of CoW operations for three different time intervals, normalized to the 2s interval. We can see that workloads such as *hpl*, *milc*, and *namd* show the highest reduction in the number of CoW operations when increasing the interval time. Compared to Fig. 1a, these workloads have the highest memory overhead. Hence, we can adjust the time interval to balance the memory and runtime overhead. For example, for *miniGhost*, employing longer time intervals reduces the runtime overhead with a relatively small impact on the memory overhead.

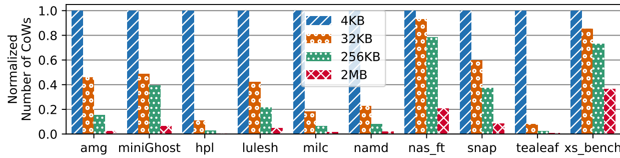


Figure 4: Normalized number of CoWs in 2s intervals

Fig. 4 shows the normalized number of CoW operations while varying the CoW granularity. We can observe that the workloads *nas_ft*, *snap*, and *xs_bench* have the lowest reduction when using the largest granularity of 2MB. Interestingly, the same workloads are also those that have the highest memory overhead according to Fig. 2a. Thus, these workloads will not benefit from increasing the granularity any further. In contrast, workloads such as *hpl* and *tealeaf* significantly benefit from increasing the granularity, since that drastically reduces the number of CoW operations while having a relatively low memory overhead. Note that the intermediate CoW granularities 32KB and 256KB also behave differently. For instance, *amg* and *miniGhost* show almost the same reduction for 32KB granularity. However, when we increase the granularity to 256KB, the number of CoW operations significantly decreases in *amg*, but remains almost the same in *miniGhost*. The reduction in *lulesh* for 256KB is between *amg* and *miniGhost*, while having almost the same reduction for 32KB. Note that we also observed the YCSB workloads to be sensitive to the CoW granularity. However, we omitted the figures because of the space limitation.

Findings (3). *The observed processing overhead of the tested applications is highly affected by the CoW granularity (additionally to the chosen snapshot interval). This confirms again that a static granularity would not be effective across all workloads.*

2.5 Spatial Locality and Temporal Locality

In Fig. 5, we visualize the impact of the CoW granularity and the time interval in combination to provide an overall view of the workloads. To do so, we arrange the workloads along two axes, namely *temporal locality* and *spatial locality*. A higher temporal locality means that the pages are accessed continuously in time intervals. As such, increasing the interval time is more beneficial since all modified pages will be copied in a smaller number of snapshots. A higher *spatial locality* means that the workload accesses a small region of the state. Therefore, employing larger CoW granularities will have more advantages. Based on the benefits of increasing the granularity and interval time, we divide the workloads into three groups.

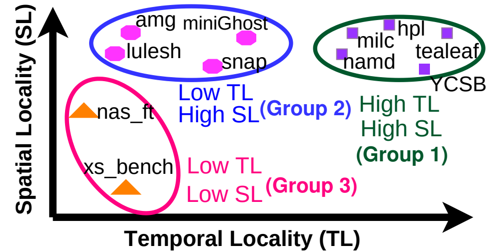


Figure 5: Spatial and temporal locality of workloads

The *first group* benefits from increasing both parameters and one can expect a significant opportunity to decrease their overheads. The *second group* has a relatively lower temporal locality but still has a high spatial locality. This group can benefit from a larger granularity. The *third group* exhibits both low temporal and low spatial locality. Therefore, optimizing both parameters is expected to have less impact on the overhead reduction.

Findings (4). *The examined applications can be partitioned into three groups with different temporal and spatial locality, based on their expected improvements when increasing the CoW granularity and/or interval time.*

3 ARCHITECTURE OF IN-SITU-COW

In Section 2, we have seen that a CoW-based snapshotting approach can indeed pay off for applications with a high spatial and temporal locality. Also, we have seen that a static approach using a fixed granularity is not sufficient. Consequently, in the following, we extend our static method of [29], which were applied therein in a prototypical way to create lightweight snapshots of main memory databases, in three ways: (1) We support arbitrary CoW granularities, as long as they are a multiple of the page size. (2) We integrate an adaptive adjustment of the CoW granularity with respect to the access pattern of the workload. (3) We present how to perform CoW asynchronously to the application's execution to reduce the processing overhead of copying modified pages. The technical details of the resulting in-situ-CoW will be presented in the following.

3.1 Implementing CoW

The core principle of in-situ-CoW consists of three parts (Fig. 6): (1) When the host application performs an allocation request, we intercept it. This allows us to serve the request with virtual memory that is backed by physical memory, to which we have a handle. (2) With the virtual and physical memory of the application under our control, we *create virtual snapshots*. (3) To ensure

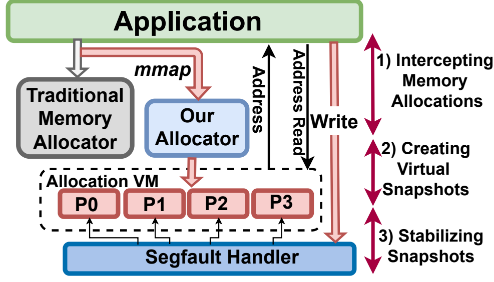


Figure 6: Basic modules in our proposed in-situ method

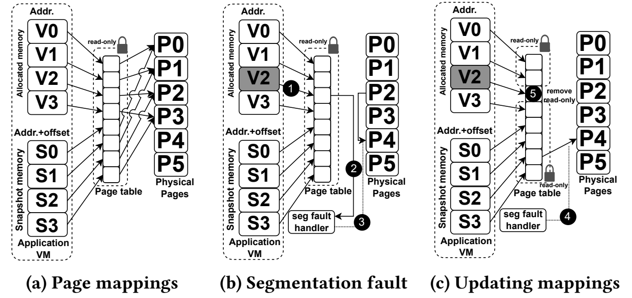


Figure 7: General workflow of our CoW

that these snapshots are *stable*, i.e., they are not changed under writes of the application, we implement a CoW mechanism. In the following, we outline the three parts in detail.

3.1.1 ***Intercepting Memory Allocations***. In order to create virtual snapshots, we require a handle to the *physical* memory of the application. Therefore, we employ function interposing to hijack the allocation process of the application. It works as follows: If an application calls a function from a dynamically linked library, the definition of the function will be resolved at runtime. Via *preloading*, however, we can *intercept* this call and ensure that our own custom definition is called instead. It allows us to intercept memory allocations and serve them using a custom allocator.

Memory allocations are served by virtual memory that is backed by anonymous memory. Since anonymous memory is not under our control, we cannot use it to create virtual snapshots. Instead, our custom allocator serves all requests by virtual memory that is backed by a so-called main-memory file, which can be created in a main-memory file system such as `tmpfs`. The key benefit of physical memory in form of a main-memory file is that we can freely map virtual memory to the physical memory at page granularity, aka memory rewiring [28]. Thus, by resolving all allocation requests of the host application by file-backed virtual memory, we essentially create a handle to the physical memory of the application.

3.1.2 ***Creating Virtual Snapshots***. By having access to physical memory, we can now create virtual snapshots (Fig. 7a). Assume we want to snapshot a virtual memory area consisting of n virtual pages $v_0, ..., v_{n-1}$, where each virtual page v_i maps to a corresponding physical page p_i of the main-memory file. We then simply create a new virtual memory area of n pages s_i and map them to p_i as well. The virtual pages $s_0, ..., s_{n-1}$ represent our virtual snapshot.

3.1.3 ***Stable Snapshots via Copy-on-Write (CoW)***. So far, our virtual snapshots are not stable since if the application writes to its memory, the changes become visible through our snapshots. To ensure the stability, we integrate the copy-on-write mechanism we showcased in [28]. After creating the virtual snapshot, we first mark all virtual pages v_i of the snapshotted virtual memory area as write-protected (Fig. 7a). As a consequence, any write to v_i by the application (❶ in Fig. 7b) will result in a *segmentation fault*. Using a custom segmentation fault handler, we now catch this segmentation fault (❷), copy the physical page mapped by v_i to an unused physical page (❸), and remap the virtual page s_i to the copy (❹ in Fig. 7c). This way, we ensure that v_i and the corresponding virtual page in the snapshot s_i map to different physical pages. We then set the protection of page v_i to *writable* (❺) and let the segmentation fault handler return. Next, the OS will now re-perform the previously failed write operation to v_i, which now definitely succeeds. Our snapshot still sees the unmodified version and thus remains stable. Fig. 7c shows the resulting state of this mechanism. Note that the detection of actual invalid memory accesses still works correctly: Our custom segmentation fault handler checks for each access whether it falls in an allocated memory area. If not, it signals to the OS to terminate the application.

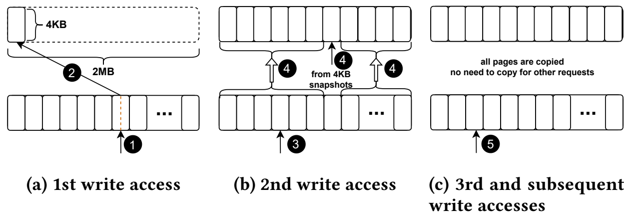
3.2 Supporting Different CoW Granularities

Of course, our method also allows to adjust the CoW granularity depending on the needs of the application. Precisely, any granularity that is a multiple of the system page size (4KB) is supported, as the remapping required in the CoW process must operate on whole pages. Further, we want to emphasize that our method is independent from the programming language in which the applications was developed, as interposing happens at the library call level.

To create virtual snapshots, we need the physical page that each virtual page is mapped to. This information is maintained by the OS and is not exposed to the user space. Thus, we maintain an auxiliary data structure, which keeps track of all mappings. We employ a B-tree and use a pre-allocated memory to get the required data structures for adding to the tree. The tree stores the user-requested mapping options, the virtual address, and the offset into the main-memory file. We employ a simple buddy allocator for our internal allocations. If the OS would provide a means to access the mapping from virtual to physical addresses, we would not need the memory-backed file and the virtual to physical mappings could have been retrieved from the OS. However, we would still need the B-tree to store the remaining metadata. Linux already provides an interface to obtain virtual to physical mappings. However, it is limited to the root user because of the security concerns. An alternative approach is to extend the OS to provide interfaces that 1) enable pointing a virtual address to the same physical address that another virtual address points to, and 2) returns virtual addresses pointing to the same physical page.

3.3 Adaptive CoW Granularity

In the previous section, we identified that the CoW granularity contains a trade-off: For a small granularity such as 4KB, each unique access to a memory page triggers a copy. While potentially, the overhead of unnecessarily copied data is low, many CoW operations must be carried out. For a larger granularity such as 2MB, potentially, the number of CoW operations decreases. However, we have the risk of copying more data unnecessarily. To make a



(a) 1st write access **(b) 2nd write access** **(c) 3rd and subsequent write accesses**

Figure 8: The workflow of adapting the CoW granularity under three requests

Table 2: Overheads of different CoW granularities. (SN: Number of segmentation faults that occur for the first *n* writes, MO: Memory overhead caused by these writes)

CoW granularity	1st Write		2nd Write		3rd Write	
	SN	MO	SN	MO	SN	MO
4KB	1	4KB	2	8KB	3	12KB
2MB	1	2MB	1	2MB	1	2MB
Adaptive (4KB/2MB)	1	4KB	2	2MB	2	2MB

concrete example for this trade-off, consider the calculation in Table 2. Imagine the application performs three write operations across a memory area of 2MB. For a static granularity of 4KB, this results in the worst case in three CoW operations copying 12KB in total where each operation triggers a segmentation fault. For a static granularity of 2MB, the three write operations trigger only a single CoW operation and respectively a single segmentation fault as processing overhead, however, 2MB must be copied in total. This means 99.5% of the copied data (509 out of 512 memory pages of size 4KB) is actually preventable overhead.

To reduce the memory overhead while still benefiting from larger granularities, we propose a variant that uses *adaptive* granularities. It works as follows: The user has the option to configure the method to a certain desired CoW granularity, such as 2MB, aka the *target granularity*. However, the first write to a 2MB area (❶ in Fig. 8) does not immediately trigger a CoW of the configured 2MB, but copies only the touched portion of 4KB (❷), aka the *initial granularity*. Only when the second write to a different memory page of that 2MB area (❸) happens, we perform a CoW of the entire 2MB (❹). As the previously copied 4KB memory region is part of the now copied 2MB region, the 4KB copy can be freed. Since the whole 2MB area is copied, all further write operations (❺) do not trigger a CoW anymore. This simple heuristic assures that we keep the number of CoW operations small for hot memory areas *while* avoiding to cause memory overhead for rarely modified areas. Note that while this strategy is rather simple and more complex strategies would be possible, it works effectively as the first write is a good indicator for successive writes. Further, our adaptive approach is capable of employing any two initial and target granularity by just changing the configuration. Of course, the optimal configuration varies between different applications. Nevertheless, we tried to present a set of configurations that work well on average. It allows users to select a configuration just by comparing the characteristics of their application with our examined applications.

In the last line of Table 2, we calculate how this adaptive strategy affects the number of segmentation faults and the memory overhead. On the first access, the adaptive strategy has the same overhead as the static 4KB granularity and hence a significantly reduced memory overhead compared to the static 2MB granularity. If the 2MB area receives a second request, then our adaptive strategy has the same memory overhead as using a static 2MB granularity, but with one additional segfault overhead. For the rest of the requests, we perform the same as the static 2MB granularity, while the number of segfaults for a granularity of 4KB continuously increases. Therefore, our adaptive method can provide a mean to balance the memory overhead and the processing overhead (i.e., the number of CoW operations respectively segmentation faults). Note that we can easily configure our proposed adaptive method to copy the whole area only on the *n*-th request.

3.4 Asynchronous CoW

As we have discussed, using an adaptive granularity reduces the amount of copied data. However, all of the modified memory pages within a time interval still need to be physically copied during the CoW operations. If the application has a large and write-heavy working set, this copying can still impose a significant performance overhead, even if adaptive granularities are used.

To reduce this overhead, we therefore propose to perform the copying *asynchronously* to the continuous execution of the application. This approach removes the need for halting the execution each time a CoW operation is performed. It works as follows:

When we create a snapshot, we can create a list of pages modified after the previous snapshot. Such pages are likely to be modified again in the next time interval. These pages are also set to read-only mode after the snapshot is created like the other pages. We start an additional thread which copies the data pages in the list into the CoW memory location pro-actively. Once a data page is copied, it is removed from the list. Note that the read-only state of the page is not changed to read-write after it is copied. This is because we still need to be notified whether this page is modified until the next snapshot. If a page is modified by the application, the segfault handler is called. It first checks if the page is in the list. If it is found, segfault handler copies the page and removes it from the list. Then similar to the other pages, it changes the state of the page into read-write and allows the application to continue its execution. Hence, the difference between modifying a page in the list and outside of it is to whether segfault handler performs data copy or not. Using this approach, copying time of a large number of pages is removed from the critical path of the application's execution.

4 EXPERIMENTAL EVALUATION

In the following section, we perform an in-depth experimental evaluation of in-situ-CoW against a set of competitive baselines. First, in Section 4.1, we evaluate the processing overhead of in-situ-CoW for different static granularities in comparison to physical snapshotting. Then, in Section 4.2, we evaluate the effect of using adaptive CoW granularities in comparison with static granularities on the memory overhead. Section 4.3 analyzes the impact of performing asynchronous CoW operations on the processing overhead. In Section 4.4, we evaluate in-situ-CoW in comparison with two oracle baselines to see how far we are away from the optimum. Section 4.5 revisits the trade-off between CoW granularity and interval time. Finally, Section 4.6 compares the behavior of in-situ-CoW against an MVCC implementation.

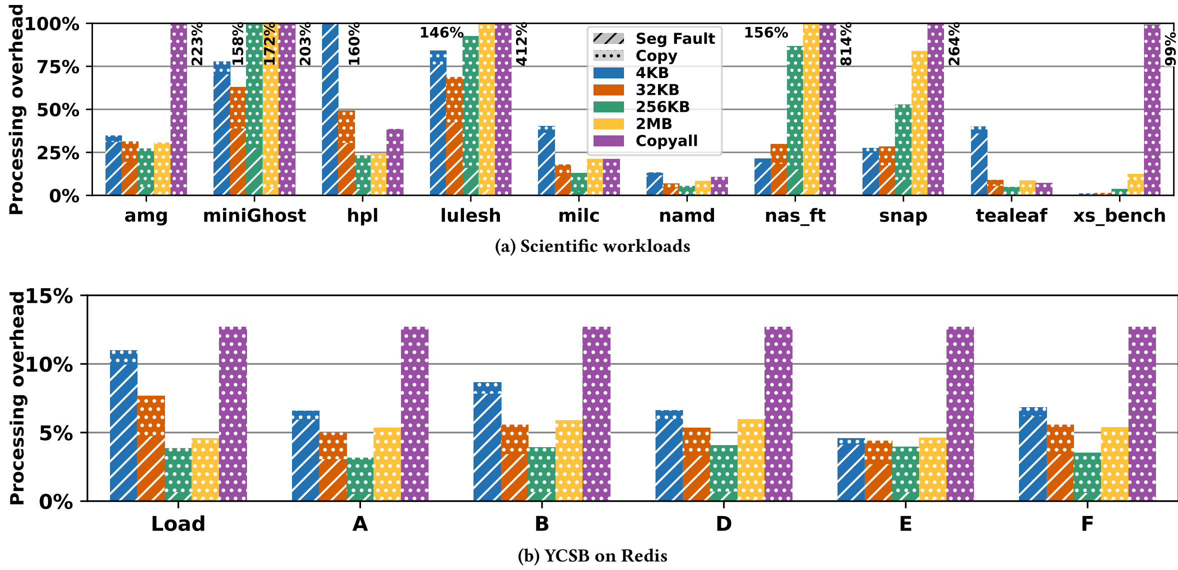


Figure 9: Processing overhead for different CoW granularities

4.1 Processing Overhead of in-situ-CoW

We start by evaluating the processing overhead of in-situ-CoW over not performing any snapshotting at all. Additionally, we show the processing overhead of *copyall*, which resembles physical snapshotting. Fig. 9 shows the results for four different statically selected CoW granularities and a time interval of 2s. Note that we show the processing overhead in relation to the time interval: For example, an overhead of 100% would mean that for each 2s interval, we need additional 2s to handle all triggered CoW operations. If the system has available CPU resources, most of this overhead will be masked. However, throughout the experiments we assumed that no idle core is available and any processing overhead will directly result in delayed application computations. To further inspect the overhead, we divide it into the overhead of handling the segmentation fault (*segfault*) including all connected processing steps and the overhead of performing the actual physical copying (*copy*).

The results show that for each workload, there exists a configuration of in-situ-CoW which significantly reduces the processing overhead in comparison to *copyall*. For most of the workloads, we can identify a configuration that results in less than 20% processing overhead, which implies that in-situ-CoW can be employed for data-intensive applications with modest overhead. Fig. 9b shows that for YCSB on Redis, our method creates a negligible overhead of less than 5% if a granularity of 256KB is used. Consequently, the fork()-based mechanism used for checkpointing would not show a smaller overhead, as it operates on 4KB memory pages. Apart from that, we can also see that for small CoW granularities, the segfault overhead dominates the copy overhead, while the copy overhead dominates the segfault overhead for larger granularities.

Comparing the results of Fig. 9 with Fig. 4 reveals that although using 2MB granularities significantly reduces the number of CoW operations, the overhead of copying larger granularities can offset the reduction of the segfault overhead. In *xs_bench* and *nas_ft*, using 4KB granularities offers the lowest performance overhead. This is in line with our classification in Fig. 5, where we estimated that these workloads will not benefit from increasing the CoW granularity. Our classification for *tealeaf* and *hpl* is also accurate, since they have the highest reduction in processing overhead when using a lager CoW granularity. In total, CoW granularities of 32KB or 256KB offer the lowest performance overhead for most of the workloads.

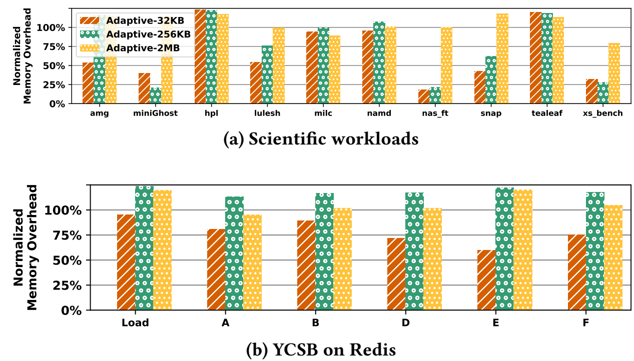
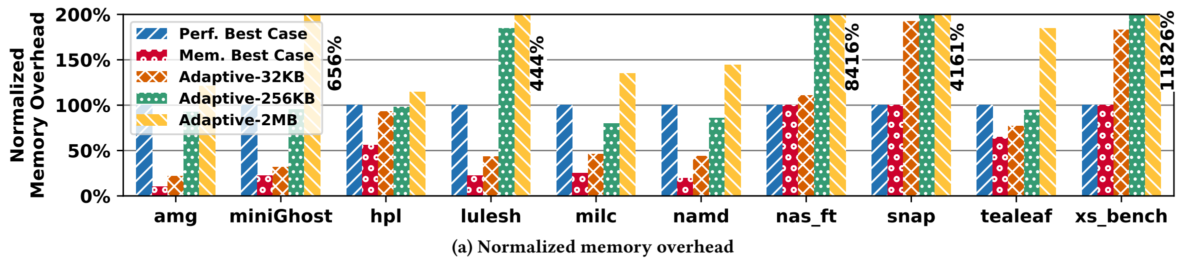


Figure 10: Normalized memory overhead for adaptive CoW granularity compared to fixed CoW granularity

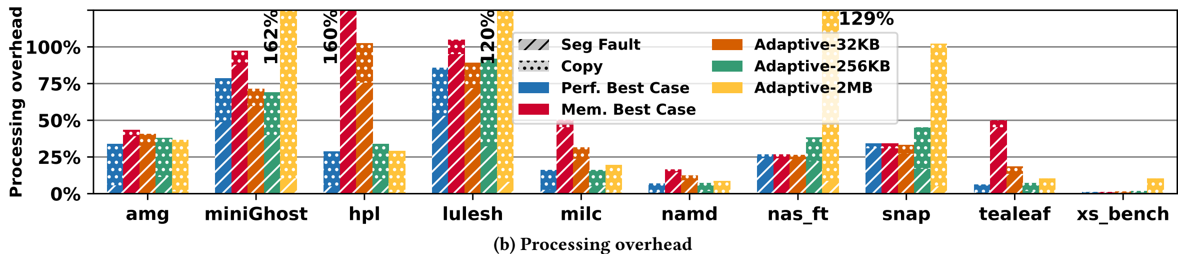
4.2 Adaptive vs Static CoW Granularity

Next, we evaluate the efficiency of the proposed adaptive adjustment of the CoW granularity. We do so in three steps:

4.2.1 Memory Overhead: One-to-one Comparison. We start by testing the effectiveness when adaptively switching from a granularity of 4KB to a 32KB, 256KB, and 2MB granularity, respectively. We normalize the memory overhead of each adaptive variant to the counterpart variant using a static granularity of the same size to get a one-to-one comparison. For instance, the results of the variant which adaptively switches from a granularity of 4KB to a granularity of 32KB are normalized to the variant using a static 32KB granularity. Note that the memory overhead



(a) Normalized memory overhead



(b) Processing overhead

Figure 11: Overhead of using an adaptive CoW granularity in comparison with the best static configurations in terms of processing overhead and memory overhead

includes all data structures needed for maintaining the snapshots as well as the space to keep the modified pages.

Fig. 10 shows the results. We can see that for six of the ten scientific workloads, adaptively setting the granularity reduces the memory overhead by more than 50%, showing the practical effectiveness of our adaptive variant. *Nas_fd* has the highest memory savings with more than 80% reduction in consumed memory. Setting the adaptive granularity to 32KB offers the highest memory savings. This is because as granularities get larger, the probability of facing a second write operating, and hence, copying the whole region, increases. *miniGhost* is the only exception, where using a granularity of 256KB results in a higher reduction. Note that an adaptive granularity of 2MB performs poorly in this workload. This shows that the memory accesses usually concentrate in areas larger than 32KB, but smaller than 256KB. In *tealeaf* and *hpl*, we observe almost no reduction of memory overhead. These workloads have a high spatial locality, and hence, rarely have regions that are seldomly accessed. The YCSB workloads exhibit a overhead reduction between around 20% and 50%. Since accesses follow a Zipfian distribution, there is a high probability that an accessed region is accessed again and hence, a large CoW is performed.

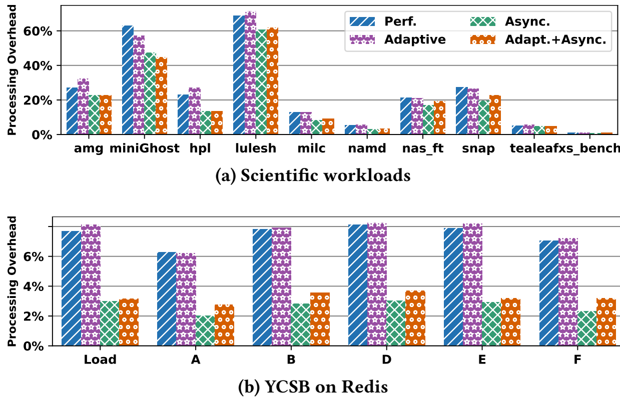
4.2.2 Memory Overhead: Comparison with Best Static CoW Granularity. In the previous section, we have seen that an adaptive CoW granularity reduces the memory overhead in comparison with the static counterpart of the same granularity. Let us see in the following how much memory overhead our adaptive approach has in comparison with the *best* static configuration. In Fig. 11a, we compare both against the best configuration in terms of processing overhead (*perf. best case*) and in memory overhead (*mem. best case*). We present the overheads normalized to *perf. best case*, while in Fig. 10 we showed the normalized overhead, compared to the corresponding static configuration. Since the adaptive approach is not very effective in YCSB workloads with a Zipfian distribution, and due to the space limitation, we omitted the YCSB results. We can see that our adaptive mechanism reduces the memory overhead by up to 78%. In comparison to *mem. best case*, which always uses a granularity of 4KB, all tested adaptive granularities unsurprisingly have a higher memory overhead. However, the memory overhead of our adaptive approach can be as low as 1.11× of the *mem. best case*. Similar to Fig. 10, a granularity of 32KB also offers the best memory overhead reduction, where both 32KB and 256KB granularities reduce the memory overhead for most of the workloads.

4.2.3 Processing Overhead: Comparison with Best Static CoW Granularity. To put the previous results into perspective, we also inspect the impact of the processing overhead. Fig. 11b shows the processing overhead of our adaptive variant in comparison with the best static configurations. We can see in the results that for all workloads, either using an adaptive 32KB granularity or a 256KB granularity shows a processing overhead close to *perf. best case*. This indicates that adaptive granularities can indeed reduce the memory overhead while causing a negligible additional processing overhead. For example, in *lulesh*, 32KB adaptive granularities impose only 4% additional processing overhead, compared to *perf. best case*. At the same time, it reduces the memory overhead by more than 57%. Similarly, in *amg*, 32KB adaptive granularities add 19% of processing overhead and in return, reduce the memory overhead by more than 88%. Further, we can see that 256KB adaptive granularities perform mostly better than the 32KB variants. In *milc* and *namd*, 256KB adaptive granularities reduce the memory overhead by more than 14% with less than 5% additional processing overhead, compared to *perf. best case*. In *miniGhost*, it further reduces the processing overhead while simultaneously reducing the memory overhead.

4.3 Asynchronous CoW

After evaluating our optimization of adaptive CoW granularity, let us evaluate the optimization of performing CoW operations asynchronously to the continuous execution of the application.

In Fig. 12, we compare the processing overhead of the best performing static configuration without asynchronous CoW (*perf.*), the best performing static configuration with asynchronous CoW (*async.*), the best adaptive configuration without asynchronous



(a) Scientific workloads

(b) YCSB on Redis

Figure 12: Processing overhead with and without asynchronous CoW

CoW (*adaptive*), and the best adaptive configuration with asynchronous CoW (*adapt. + async.*). As we can see in Fig. 12a, the performance improvement of asynchronous CoW varies between the scientific workloads. When using a fixed CoW granularity, we observe up to 45% reduction of processing overhead (26% on average) when using asynchronous CoW. This performance improvement trend is mostly maintained when we switch from static to adaptive CoW granularities, by up to 50% of improvement (23% on average). This shows that regardless of the CoW granularity and whether a static or adaptive approach is used, asynchronous CoW significantly reduces the processing overhead. Further, since this optimization does not impose any additional memory overhead, it should be applied to all configurations and workloads.

In YCSB workloads, the async. approach significantly reduces the performance overhead. In all examined workloads, this overhead is reduced by more than 60% which shows the effectiveness of our async. approach. Note that we employed the optimal granularity for async. (i.e., 256KB). Simply using the async. approach without our adaptive granularity selection will not yield in such a performance overhead reduction.

Additionally, we have examined the impact of varying the number of threads when running an application for the application *miniGhost*. Therein, the performance of async. CoW is only slightly degraded when increasing the thread count from 8 to 16, 32, and 64 by 2.1%, 4.2%, and 4.3%.

4.4 Comparison against Oracle Baselines

To evaluate how much overhead in-situ-CoW has in comparison to the optimum, we introduce two optimal CoW strategies, where all accesses in the interval are known beforehand. As a consequence, the CoW strategy can set the optimal granularity for individual memory regions. Note that the maximum granularity that can be set by the optimal CoW strategies is still 2MB, as larger ones did not result in further improvements, as we identified experimentally.

Our *first* implementation is called *Optimal Memory CoW* strategy (**OM-CoW**), as it causes the minimal memory overhead. To achieve this, the CoW granularity of this mechanism is set to 4KB. Note that if several continuous 4KB pages are accessed in the same time interval, all of them are copied in a single CoW operation to minimize the processing overhead.

Our *second* implementation is called *Optimal Performance CoW* (**OP-CoW**), as it selects for each 2MB memory region the optimal

Algorithm 1: OP-CoW: Optimal Performance CoW

```
   Data: all_accessed_pages, proc_overhead
1  foreach 2MB chunk do
2  |   p_cost, m_cost ← find_optimal(chunk.start, 9) /* 2^9 pages */
3  end
4  Function find_optimal(start, depth):
5  |   if depth == 0 then
6  |   |   if start ∈ all_accessed_pages then
7  |   |   |   return (proc_overhead[1],1);
8  |   |   else
9  |   |   |   return (0,0) /* (CPU, Memory) overheads */
10 |   |   end
11 |   end
12 |   curr_overhead ← proc_overhead[2^depth];
13 |   l_cost, l_cost_mem ← find_optimal(start, depth-1);
14 |   r_cost, r_cost_mem ← find_optimal(start+2^(depth-1), depth-1);
15 |   if l_cost+r_cost < curr_overhead then
16 |   |   return
   |   |    (l_cost + r_cost, left_cost_mem + right_cost_mem);
17 |   else
18 |   |   return (curr_overhead, 2^depth);
19 |   end
```

CoW granularity. It recursively calculates the performance cost of using two small adjacent region or one larger region containing both of them as CoW granularity. For the calculation, we require two inputs: a) Accessed 4KB pages in the time interval which we obtain from the memory traces. b) The processing cost of performing a CoW operation for all possible granularities. Algorithm 1 provides the pseudo-code for the generation of **OP-CoW**. For each 2MB block, we call `find_optimal()` and accumulate the processing and memory overheads. `find_optimal()` recursively calculates the cost of using a smaller granularity and decides whether employing a larger granularity can reduce the processing overhead. Line 5 to line 11 show the exit condition. We check whether the data page is accessed in this time interval or not. If it is accessed, then performing CoW on this page requires a processing overhead for one page and also occupies one memory page. If not accessed, no overheads are imposed. Line 13 and line 14 split the current granularity into two smaller granularities and call the same function for both. Line 15 to line 19 decide if the cost of using the current granularity is lower or splitting it can reduce the processing overhead.

Additionally, we include a third baseline **OP-Traditional**, which shows the performance of using physical snapshotting. Precisely, OP-Traditional has knowledge of all accesses in the future and therefore physically copies all 4KB pages that will be modified until the next snapshot in one go.

Figure 13a shows the results in terms of processing overhead. Aligning these with the classification of workloads in Fig. 5, we can see that in-situ-CoW shows only a slight overhead over OP-CoW for workloads with a high temporal and spatial locality. For these workloads, it also performs better or as good as OP-Traditional. However, we can also see that for workloads with low temporal locality but high spatial locality, as expected, in-situ-CoW shows overhead over the performance oriented oracle baselines. Further, in comparison to OM-CoW and the best static CoW configuration in terms of memory footprint, we can see that in-situ-CoW has a significantly smaller processing overhead in nearly all cases.

In Fig. 13b, we shift the focus on the memory overhead of in-situ-CoW, where we show all results normalized to in-situ-CoW. The best memory case and OM-CoW both employ a granularity

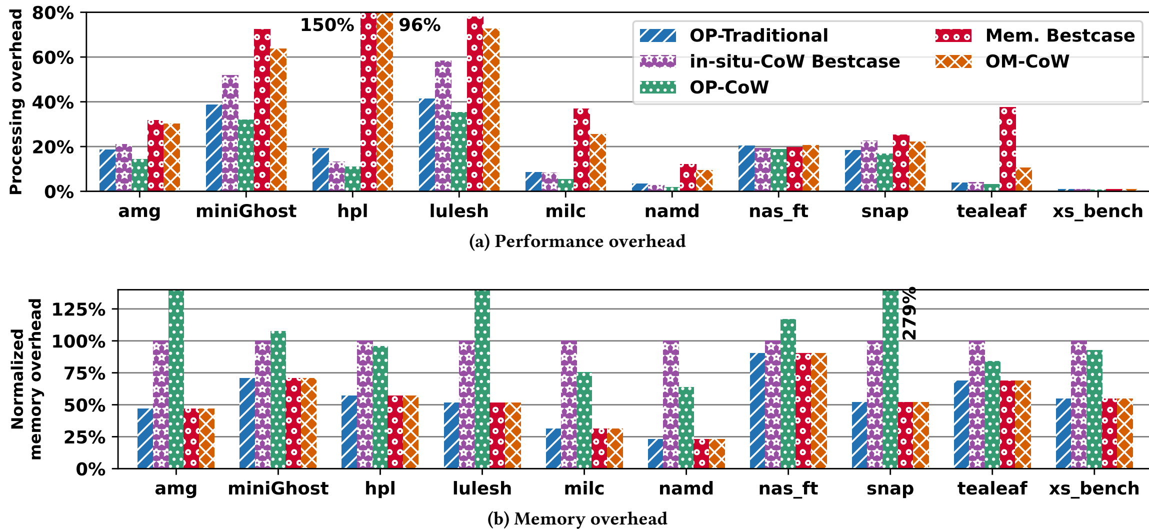


Figure 13: Comparison of in-situ-CoW against baselines

of 4KB, and hence, have the same overhead. We can see that unfortunately for no workload, in-situ-CoW can reach the memory overhead of the memory optimal case and of OM-CoW. However, under several workloads, in-situ-CoW shows a lower memory overhead than OP-CoW. As performance is our main optimization target, this shows the effectiveness of our approach.

4.5 CoW Granularity vs Interval Time

Finally, we investigate the trade-off between the CoW granularity and the interval time. The chosen interval time depends primarily on the use-case, i.e., how often a new snapshot is required by the user. However, as the cost of creating a snapshot varies across applications, it must be factored in when setting the interval time.

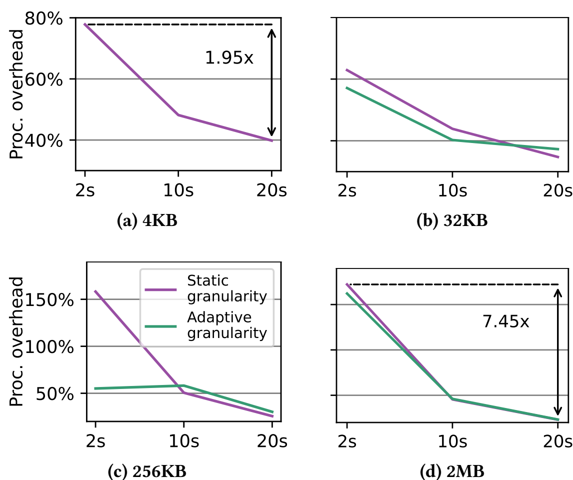


Figure 14: Performance tradeoff for *miniGhost* workload

In Fig. 14, we vary the interval time and report the processing overhead for the *miniGhost* workload. For the static approaches, increasing the interval time reduces the processing overhead under all CoW granularities. However, the amount of reduction is significantly varies. For instance, the overhead when going from an interval time from 2s to 20s is reduced by 1.95× for 4KB granularities, while it is reduced by 7.45× for 2MB granularities. Thus, for larger CoW granularities, increasing the interval time has a higher impact on the reduction of processing overhead.

Further, we also measured the overhead for the adaptive approach. We can see that by increasing the interval time, the overhead of the adaptive method is reduced at a lower pace. This is because for longer intervals, there is a higher chance that at least two pages are accessed in a granule, and hence, the adaptive method imposes a slightly higher overhead than the static counterpart.

4.6 Virtual Snapshotting vs MVCC

Apart from physical snapshotting, multi-version concurrency control (MVCC), which is implemented by a wide range of modern DBMSs [6, 14, 21, 37] to coordinate the execution of concurrent transactions, is another option to create snapshots. The core idea is simple: On updates, instead of overwriting the old version of an entry with the new one, MVCC preserves the old version in a separate version chain. This allows to execute potentially long-running queries on a consistent version of the database (aka a snapshot) *while* being able to execute modifying transactions concurrently.

To analyze the performance differences between MVCC and virtual snapshotting, we perform the following benchmark executing four phases: After allocating an integer array of 400MB, we (1) perform a certain amount of uniformly distributed updates. Then, (2) we take a snapshot. After that, (3) we perform another batch of updates, which will not be reflected by the snapshot. Finally, (4) we iterate over snapshot entries and sum them up.

We run MVCC, which orders versions from newest (in-place) to oldest (deepest in the version chain), with dynamically allocated entries and with pre-allocated entries (each entry consists of the integer value, an 8B timestamp, and a pointer to the next entry). While taking an MVCC snapshot requires only of keeping a timestamp, accessing an entry of the snapshot requires the traversal of the corresponding version chain from newest to oldest until the first version is found that can be seen by the

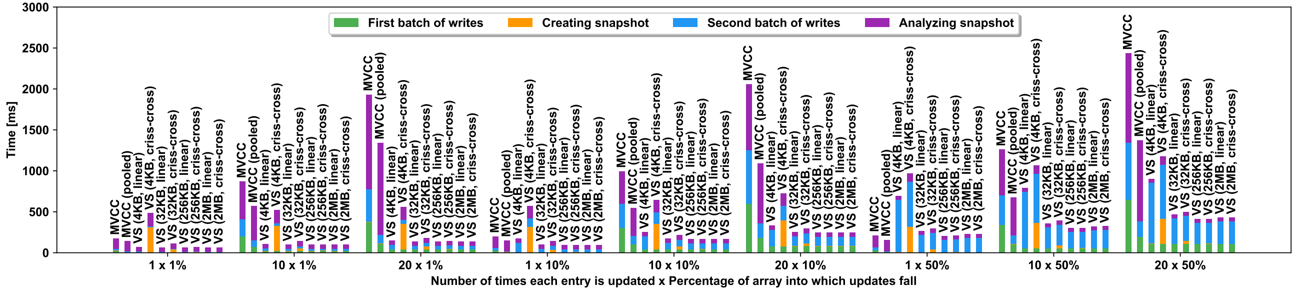


Figure 15: Comparing virtual snapshotting with MVCC for 1% of entries being updated.

snapshot. For virtual snapshotting, we test the previously used four different CoW granularities and consider both the best case, where the virtual snapshot consists of a single linear mapping, as well as the worst case, where all pages of the snapshot are mapped in a criss-cross fashion.

Figure 15 shows the results for 1% of the entries being updated. We vary the range from which these 1% entries are picked, where we test 1%, representing very high skew, and 10% as well as 50% representing lower skew. Further, we vary the number of times each entry is updated, from only once over 10 times to 20 times. We observe that the performance of MVCC depends on the amount of updates much more than in-situ-CoW. While for only one update per selected entry, MVCC is competitive to in-situ-CoW, for multiple updates per entry, the performance of MVCC deteriorates due to the build up of longer version chains. This makes especially the analysis of the snapshot expensive, which is cheap and independent of the number of updates for in-situ-CoW. In summary, virtual snapshotting offers the a more workload-robust performance – while not requiring to modify the applications in any way.

4.7 Lessons Learned

Based on the presented experimental results, let us summarize the most important results and lessons learned:

(1) Our 16 different applications/workloads have different requirements for a CoW-based snapshotting mechanism. However, it is (a) possible to categorize them in terms of locality of the access pattern, and (b) in-situ-CoW can be configured to work well for a large number of workloads.

(2) Our in-situ-CoW shows a significantly lower processing overhead than both physical snapshotting and MVCC. Against MVCC, in-situ-CoW shows a significantly smaller cost for using the snapshot, as no reconstruction is required whatsoever. In summary, for fourteen out of our 16 workloads, the overhead of in-situ-CoW over not creating snapshots snapshotting is less than 30%.

(3) There is no single CoW granularity that works best for all workloads. However, the granularities 32KB and 256KB have the smallest processing overhead for fourteen out of 16 workloads, showing that CoW using the system page granularity is not optimal.

(4) It pays off to use an adaptive CoW granularity. The memory overhead over the counterpart configuration was reduced to 50% for six out of 16 workloads. Over the best static CoW granularity, the adaptive strategy reduces the overhead by up to 78%, where its memory overhead is only 11% higher than the static configuration with the lowest memory overhead. In terms of processing overhead, when using target granularities of 32KB and 256KB, the adaptive strategy performs comparable to the static configuration with the lower processing overhead.

(5) It pays off to perform CoW asynchronously. For the scientific workloads, copying asynchronously improves the processing overhead by up to 50%. For the YCSB workload, it even improves by up to 60%. Further, asynchronous copying is only negligibly impacted by the number of used threads of the host application: When doubling the number of threads, the performance degrades by less than 4%.

(6) Our in-situ-CoW gets close to the performance-optimal oracle baseline in terms of processing-overhead, if the spatial and temporal locality of the workload is high. In terms of memory-overhead, the memory-optimal oracle baseline cannot be surpassed.

(7) CoW granularity and snapshotting time interval trade off performance overhead. The larger the CoW granularity, the higher is the reduction in processing overhead when increasing the time interval.

5 ANALYSIS WORKFLOW

After discussing and evaluating how virtual snapshots can be created efficiently, let us now discuss how these snapshots could actually be used for in-situ analysis. Note that using our in-situ-CoW method, this in-situ analysis could be performed on DBMSs, which do not provide an internal snapshotting mechanism, as well as on HPC applications.

5.1 In-situ Frameworks

To perform the actual in-situ analysis, there exist various frameworks [4, 12, 13, 31, 33, 35]. For instance, ADIOS [13] provides an API that allows host applications to expose their current state to it. A module called Flexpath [9] then creates a physical snapshot of the state and transfers it to analysis jobs. Fig. 16a visualizes the principle. While this approach is simple, the downside is that the amount of required memory is doubled and the application must be halted during the copying.

Note that it is also possible to transfer only the modified parts of the state to ADIOS, as shown in Fig. 16b. However, to enable this, the host application should be heavily adapted to partition the data in order to create clusters of modified data. Further, the application needs to mark the modified parts explicitly. This becomes cumbersome to infeasible in complex scientific applications, especially if data is also modified by third-party libraries.

By integrating in-situ-CoW into ADIOS, which is left as future work, these limitations could be fully resolved. Additionally, our approach enables the asynchronous copying of the modified data to the in-situ module, while the application an continues to run.

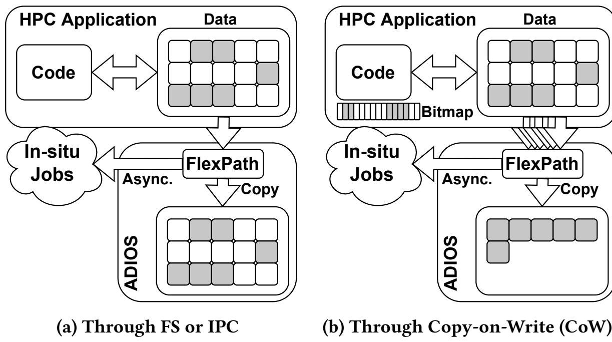


Figure 16: Data flow between processing and analysis jobs

5.2 Interpreting Snapshots

After a snapshot has been transferred to the in-situ framework, it must be interpreted before any analysis can happen. This step is independent from whether the snapshot has been created traditionally or by using in-situ-CoW. The interpretation must be provided in form of type information, which is used by the framework to cast the raw memory snapshot into a a well-formatted layout. For HPC applications, the produced state is often structured using some standard format such as NetCDF or HDF5, which eases and generalizes this step. If no standard format is used, the analysis job has to provide the type information. We want to emphasize that providing this type information (by having internal knowledge about the state management) is required for all alternative approaches as well. However, by using in-situ-CoW, we at least avoid the modification of the host application for creating snapshots, which might be complex or even infeasible.

5.3 Analyzing Snapshots

After the snapshot has been interpreted, the actual analysis jobs can be executed. In scientific environments, where applications are performing long-running computations, these jobs typically fall into the categories of either (a) progress monitoring (how long does it still take to finish computation?), (b) error tracking (is the computation still behaving as intended?), or (c) extracting intermediate results, such as an approximation of the final result. Apart from that, in-situ analysis jobs could be used to support lifecycle monitoring providing information on the health or load of the system for applications that natively do not support it (similar to RedisInsight or Oracle AWR, but without deep integration into the host system).

5.4 Managing Snapshots

Note that in-situ-CoW allows multiple analysis jobs to create and process multiple virtual snapshots simultaneously. From the perspective of the analysis job, its virtual snapshot is completely independent from other existing virtual snapshots, although they might share physical pages. Consequently, there is no coordination required between jobs required.

As soon as a virtual snapshot is not of use anymore, it can be deleted. When garbage-collecting a snapshot, we ensure to only free those pages that are not shared with any newer snapshot. This is the case for all pages that have not been modified by the application after taking the snapshot.

6 RELATED WORK

Apart from the obvious related work of [28] and [29], there exists other related work in the field worth discussing.

Regarding virtual snapshotting, Redis was not the first system to implement fork()-based snapshotting. The HyPer database system [16] introduced the technique of processing spawning to create consistent snapshots for running OLAP queries in the child processes, while modification OLTP transactions were executed in the main process. However, due to the previously mentioned downsides, the technique was later swapped with a traditional MVCC implementation. To overcome the limitations of fork(), in [30], a new system call was developed to essentially create virtual memory snapshots just like fork() but within the process. While this solves the aforementioned problem, it requires heavy OS changes. SwingDB [20] follows a similar approach but directly modifies the virtual memory subsystem of the Linux kernel.

In the database world, MVCC has become the de-facto standard of concurrency control [6, 14, 21, 27, 37]. However, it has downsides in comparison to our approach: (a) It must be deeply integrated in the engine. For example, our tested applications do not perform any multi-versioning. Hence, using it to create snapshots requires rewriting these applications. In contrast, our approach is designed to be as minimally invasive to the application logic as possible. (b) In MVCC, a snapshot must be carefully constructed from the individual versions by traversing version chains, which is a costly process we can avoid.

Saving the state of the application during runtime also has other use-cases. For instance, checkpointing is used in long-running scientific jobs to reduce the cost of failures. The probability of failure increases in long runs. Checkpoint-restart is therefore used in such scientific jobs to save the state in time intervals so that it can restart from the saved checkpoints [11]. Checkpointing requires halting the execution across all nodes, saving the state, and then resuming. In the in-situ analysis, the state is analyzed immediately, and hence, can be kept in the main memory. On contrary, the state in the checkpoint-restart needs to be saved on a storage device.

7 CONCLUSION

In this work, we performed an extensive experimental study on the usefulness of in-situ based virtual snapshotting on 16 representative applications and workloads from the HPC and data management domain. As basis, we used our method in-situ-CoW, an in-situ virtual snapshotting mechanism that does neither halt the host application during snapshot creation nor does it require a deep modification of it. We extended in-situ-CoW to support arbitrary CoW granularities, adaptive switching of CoW granularities, as well as asynchronous copying. In comparison to traditional physical snapshotting, in-situ-CoW reduces the performance overhead by up to 98% (66% on average) for scientific workloads and by up to 64% (45% on average) for YCSB. In comparison to MVCC, in-situ-CoW reduces the performance overhead by up to 89% (25% on average) under write-intensive workloads.

REFERENCES

[1] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. 2006. The landscape of parallel computing research: A view from berkeley. (2006).

[2] David Bailey, Tim Harris, William Saphir, Rob Van Der Wijngaart, Alex Woo, and Maurice Yarrow. 1995. *The NAS parallel benchmarks 2.0.* Technical Report. Technical Report NAS-95-020, NASA Ames Research Center.

[3] Richard Frederick Barrett, Michael Allen Heroux, and Courtenay Thomas Vaughan. 2012. MiniGhost : a miniapp for exploring boundary exchange strategies using stencil computations in scientific parallel computing. (4 2012).

[4] Andrew C. Bauer, Berk Geveci, and Will Schroeder. 2018. *ParaView Catalyst User's Guide.* Kitware Inc.

[5] Claude Bernard, Tom Burch, Tom DeGrand, Carleton DeTar, Steve Gottlieb, Urs Heller, James Hetrick, Craig McNeile, Kostas Orginos, , James Osborn, Kari Rummukainen, Bob Sugar, and Doug Toussaint. 2002. *MILC.* https://web.physics.utah.edu/~detar/milc/milcv6.html accessed 10. Mar. 2022.

[6] Jianjun Chen, Yonghua Ding, Ye Liu, Fangshi Li, Li Zhang, Mingyi Zhang, Kui Wei, Lixun Cao, Dan Zou, Yang Liu, Lei Zhang, Rui Shi, Wei Ding, Kai Wu, Shangyu Luo, Jason Sun, and Yuming Liang. 2022. ByteHTAP: ByteDance's HTAP System with High Data Freshness and Strong Data Consistency. *Proc. VLDB Endow.* 15, 12 (2022), 3411–3424. https://doi.org/10.14778/3554821.3554832

[7] Jieyang Chen, Qiang Guan, Zhao Zhang, Xin Liang, Louis James Vernon, Allen McPherson, Li-Ta Lo, Patricia Grubel, Tim Randles, Zizhong Chen, and James P. Ahrens. 2018. BeeFlow: A Workflow Management System for In Situ Processing across HPC and Cloud Systems. In *38th IEEE International Conference on Distributed Computing Systems (ICDCS), Vienna, Austria.* 1029–1038.

[8] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC), Indianapolis, Indiana, USA, June 10-11*, Joseph M. Hellerstein, Surajit Chaudhuri, and Mendel Rosenblum (Eds.). 143–154. https://doi.org/10.1145/1807128.1807152

[9] Jai Dayal, Drew Bratcher, Greg Eisenhauer, Karsten Schwan, Matthew Wolf, Xuechen Zhang, Hasan Abbasi, Scott Klasky, and Norbert Podhorszki. 2014. Flexpath: Type-Based Publish/Subscribe System for Large-Scale Science Analytics. In *14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), Chicago, IL, USA, May 26-29.* 246–255. https://doi.org/10.1109/CCGrid.2014.104

[10] Arnaldo Carvalho De Melo. 2010. The new linux perf tools. In *Linux Kongress.*

[11] Alvaro Frank, Manuel Baumgartner, Reza Salkhordeh, and André Brinkmann. 2021. Improving checkpointing intervals by considering individual job failure probabilities. In *35th IEEE International Parallel and Distributed Processing Symposium (IPDPS), May 17-21.* 299–309.

[12] Yuankun Fu, Feng Li, Fengguang Song, and Zizhong Chen. 2018. Performance analysis and optimization of in-situ integration of simulation with data analysis: zipping applications up. In *International Symposium on High-Performance Parallel and Distributed Computing (HPDC), June 11-15.*

[13] William F. Godoy, Norbert Podhorszki, Ruonan Wang, Chuck Atkins, Greg Eisenhauer, Junmin Gu, Philip E. Davis, Jong Choi, Kai Germaschewski, Kevin A. Huck, Axel Huebl, Mark Kim, James Kress, Tahsin M. Kurç, Qing Liu, Jeremy Logan, Kshitij Mehta, George Ostrouchov, Manish Parashar, Franz Poeschel, David Pugmire, Eric Suchyta, Keichi Takahashi, Nick Thompson, Seiji Tsutsumi, Lipeng Wan, Matthew Wolf, Kesheng Wu, and Scott Klasky. 2020. ADIOS 2: The Adaptable Input Output System. A framework for high-performance data management. *SoftwareX* 12 (2020), 100561.

[14] Martin Grund, Philippe Cudré-Mauroux, and Samuel Madden. 2011. A Demonstration of HYRISE - A Main Memory Hybrid Storage Engine. *Proc. VLDB Endow.* 4, 12 (2011), 1434–1437. http://www.vldb.org/pvldb/vol4/p1434-grund.pdf

[15] Ian Karlin, Abhinav Bhatele, Jeff Keasler, Bradford L. Chamberlain, Jonathan D. Cohen, Zachary DeVito, Riyaz Haque, Dan Laney, Edward Luke, Felix Wang, David F. Richards, Martin Schulz, and Charles H. Still. 2013. Exploring Traditional and Emerging Parallel Programming Models Using a Proxy Application. In *27th IEEE International Symposium on Parallel and Distributed Processing (IPDPS), Cambridge, MA, USA, May 20-24.* 919–932.

[16] Alfons Kemper and Thomas Neumann. 2011. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *Proceedings of the 27th International Conference on Data Engineering (ICDE), April 11-16.* 195–206.

[17] Lawrence Livermore National Laboratory. 2019. *Algebraic multigrid benchmark.* Retrieved March 10, 2022 from https://github.com/LLNL/AMG

[18] Los Alamos National Security. 2020. SNAP. https://github.com/lanl/SNAP accessed 17. Mar. 2022.

[19] Simon McIntosh-Smith, Matthew Martineau, Tom Deakin, Grzegorz Pawelczak, Wayne P. Gaudin, Paul Garrett, Wei Liu, Richard P. Smedley-Stevenson, and David Beckingsale. 2017. TeaLeaf: A Mini-Application to Enable Design-Space Explorations for Iterative Sparse Linear Solvers. In *IEEE International Conference on Cluster Computing (CLUSTER), September 5-8.* 842–849.

[20] Qingzhong Meng, Xuan Zhou, Shiping Chen, and Shan Wang. 2016. SwingDB: An Embedded In-memory DBMS Enabling Instant Snapshot Sharing *(Lecture Notes in Computer Science, Vol. 10195).* 134–149.

[21] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. 2015. Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives (Eds.). ACM, 677–689. https://doi.org/10.1145/2723372.2749436

[22] Gagandeep Panwar, Da Zhang, Yihan Pang, Mai Dahshan, Nathan DeBardeleben, Binoy Ravindran, and Xun Jian. 2019. Quantifying Memory Underutilization in HPC Systems and Using it to Improve Performance via Architecture Support. In *Proceedings of the 52nd IEEE/ACM International Symposium on Microarchitecture (MICRO), October 12-16.* 821–835.

[23] Konstantinos Parasyris, Giorgis Georgakoudis, Leonardo Bautista-Gomez, and Ignacio Laguna. 2021. Co-Designing Multi-Level Checkpoint Restart for MPI Applications. In *21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid), Melbourne, Australia, May 10-13.* 103–112.

[24] Arnab Kumar Paul, Olaf Faaland, Adam Moody, Elsa Gonsiorowski, Kathryn Mohror, and Ali Raza Butt. 2020. Understanding HPC Application I/O Behavior Using System Level Statistics. In *27th IEEE International Conference on High Performance Computing, Data, and Analytics (HiPC), Pune, India, December 16-19.* 202–211.

[25] A. Petitet, R. C. Whaley, J. Dongarra, and A. Cleary. 2018. *HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers.* http://www.netlib.org/benchmark/hpl [accessed 10. Jul. 2022].

[26] James C. Phillips, Yanhua Sun, Nikhil Jain, Eric J. Bohm, and Laxmikant V. Kalé. 2014. Mapping to Irregular Torus Topologies and Other Techniques for Petascale Biomolecular Simulation. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC), Nov. 16-21.*

[27] Dan R. K. Ports and Kevin Grittner. 2012. Serializable Snapshot Isolation in PostgreSQL. *Proc. VLDB Endow.* 5, 12 (2012), 1850–1861. https://doi.org/10.14778/2367502.2367523

[28] Felix Martin Schuhknecht, Jens Dittrich, and Ankur Sharma. 2016. RUMA has it: Rewired User-space Memory Access is Possible! *Proc. VLDB Endow.* 9, 10 (2016), 768–779.

[29] Felix Martin Schuhknecht, Aaron Priesterroth, Justus Henneberg, and Reza Salkhordeh. 2021. AnyOLAP: Analytical Processing of Arbitrary Data-Intensive Applications without ETL. *Proc. VLDB Endow.* 14, 12 (2021).

[30] Ankur Sharma, Felix Martin Schuhknecht, and Jens Dittrich. 2018. Accelerating Analytical Processing in MVCC using Fine-Granular High-Frequency Virtual Snapshotting. In *Proceedings of the International Conference on Management of Data (SIGMOD), June 10-15.* 245–258.

[31] Vítor Silva, Daniel de Oliveira, Marta Mattoso, and Patrick Valduriez. 2018. DfAnalyzer: Runtime Dataflow Analysis of Scientific Applications using Provenance. *Proceedings of the VLDB Endow.* 11, 12 (2018), 2082–2085. https://doi.org/10.14778/3229863.3236265

[32] Jonathan M. Smith and Gerald Q. Maguire Jr. 1988. Effects of Copy-on-Write Memory Management on the Response Time of UNIX Fork Operations. *Computing Systems* 1, 3 (1988), 255–278. http://www.usenix.org/publications/compsystems/1988/sum_smith.pdf

[33] Devesh Tiwari, Simona Boboila, Sudharshan S. Vazhkudai, Youngjae Kim, Xiaosong Ma, Peter Desnoyers, and Yan Solihin. 2013. Active flash: towards energy-efficient, in-situ data analytics on extreme-scale machines. In *11th USENIX conference on File and Storage Technologies (FAST), February 12-15.* 119–132.

[34] John R Tramm, Andrew R Siegel, Tanzima Islam, and Martin Schulz. 2014. XSBench - The Development and Verification of a Performance Abstraction for Monte Carlo Reactor Analysis. In *The Role of Reactor Physics toward a Sustainable Future (PHYSOR).* Kyoto.

[35] Venkatram Vishwanath, Mark Hereld, Vitali A. Morozov, and Michael E. Papka. 2011. Topology-aware data movement and staging for I/O acceleration on Blue Gene/P supercomputing systems. In *Conference on High Performance Computing Networking, Storage and Analysis (SC), November 12-18.*

[36] Zhe Wang, Pradeep Subedi, Matthieu Dorier, Philip E. Davis, and Manish Parashar. 2021. Adaptive Placement of Data Analysis Tasks For Staging Based In-Situ Processing. In *28th IEEE International Conference on High Performance Computing, Data, and Analytics (HiPC), Bengaluru, India, December 17-20.*

[37] Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. 2017. An Empirical Evaluation of In-Memory Multi-Version Concurrency Control. *Proc. VLDB Endow.* 10, 7 (2017), 781–792. https://doi.org/10.14778/3067421.3067427