LDT: Lightweight Dirty Tracking of Memory Pages for x86 Systems

Rohit Singh, Arun KP, Debadatta Mishra
Department of Computer Science and Engineering
Indian Institute of Technology Kanpur, India
{rsingh, kparun, deba}@cse.iitk.ac.in

***Abstract*—Incremental memory checkpointing is a crucial primitive required by applications such as live migration, cloning, debugging etc. In many implementations of incremental checkpointing, the memory modifications are tracked by restricting write access to memory pages using the support provided in the memory management unit (MMU) hardware. Disabling write access impacts the performance of applications because of the page faults induced in the form of permission violation on memory store operations by the applications.**

In this paper, we propose LDT, a light-weight memory write monitoring mechanism to support efficient incremental checkpointing. LDT is designed to work in systems with MMU support for page dirty indicators (such as dirty-bit in x86 systems) by enabling polymorphic use of the indicators such that no other subsystem is impacted because of LDT. We design and implement LDT in the Linux kernel as an alternate to the existing write-restriction based technique. We establish the correctness and comparative efficiency of LDT through extensive experimental analysis. The results show that under write-heavy workloads, LDT outperforms write-restriction based technique by a factor of 2x in execution time. For real-world workload benchmarks such as Redis, LDT results in 2% to 8% throughput improvement compared to the state-of-the-art dirty tracking technique.

***Index Terms*—operating system, memory management, page modification monitoring**

I. Introduction

Incremental checkpointing [1]–[3] of the execution state of applications encapsulated as processes, containers or Virtual Machines (VMs) is shown to be very useful. Live migration of containers and VMs [4], [5], fault tolerance and high availability [1], [6], application debugging using record and replay [7], fast initialization [8], and memory rejuvenation [9] are some example solutions built using the incremental checkpointing mechanisms as the primary enabler. Furthermore, with non-volatile memory (NVM) systems, the requirement for periodic checkpointing becomes more crucial to support application/OS level checkpointing and crash recovery [10], [11]. Incremental checkpointing involves capturing and saving the state changes of an execution entity (process, container or VM) across multiple checkpoint intervals such that the checkpoint size is minimized resulting in efficient checkpointing-based solutions. For example, live migration of containers/VMs with incremental checkpointing reduces the data transfer overheads, providing efficient migration capabilities whereas reduced checkpoint size in an NVM system reduces the periodic-saving overhead of the execution state to the NVM.

Tracking changes to the *memory state* of an application (a.k.a. dirty tracking) across consecutive checkpoint intervals is one of the significant operations to support incremental checkpointing. As opposed to copy overheads at the end of any checkpoint or tracking interval, dirty tracking interferes with the application execution in a continuous manner for the whole duration. For example, write fault based dirty tracking for any application, when enabled, would introduce additional page faults resulting in degraded application performance (Section II). Furthermore, the copy overhead can be significantly reduced by the usage of fast persistent storage devices (e.g., NVMs) increasing the emphasis on availability of efficient dirty tracking techniques. The objective of this paper is to propose an efficient technique to *reduce the overheads* associated with state-of-the-art incremental dirty tracking techniques.

For tracking changes to the *memory state* of an application, conventional systems allow tracking at page-level granularity, leveraging the access permission feature available in the memory management unit (MMU) hardware. One such common approach is forcing a page-fault on the first write to a page by removing write permission from the translation entry (e.g., the page table entry or PTE in x86) [12]. A typical incremental memory dirty tracking technique removes the write permission bit in the page table at the start of the checkpoint interval for the tracked memory areas. This results in a page-fault on first write to any address in the tracked memory area during the interval; an associated tracking mechanism can record the corresponding pages as dirty. The overheads due to the page-faults introduced because of dirty tracking is significant as we show in Section II-C. Two potential factors contributing to the performance degradation of a tracked application are—*(i)* page-fault handling overheads, and *(ii)* TLB misses due TLB invalidation while changing access permissions of dirty tracked pages. There are also additional overheads at the start of the checkpoint (to walk the page table to change the permissions) and while consuming the dirty tracked information from the user space at the end of checkpoint interval. For example, write-fault based dirty tracking with `SoftDirty` feature in the Linux OS [13] requires the tracker application to read and process the `pagemap` file to find the dirtied pages at the end of a checkpoint interval.

A possible approach to track the memory modifications at a page granularity could be using the dirty-bit set by

the MMU hardware during address translation. On write to a virtual address, the page-table walker hardware (part of MMU) sets the dirty bit in the corresponding page table entry (PTE), if the dirty-bit is not set already. A dirty-bit based tracking mechanism can clear the dirty-bit in PTEs at the start of a tracking interval and inspect the dirty bit at the end to record dirtied pages. Compared to the write-fault based tracking, the dirty-bit based mechanism could result in significant performance advantage by avoiding page-faults on first writes. However, designing a dirty-bit based dirty tracking mechanism in conventional operating system such as Linux presents some non-trivial challenges. First, different OS subsystems such as the file system, page swapper etc. may depend on the dirty-bit in PTE to ensure their correct functioning. Therefore, out-of-turn clearing of the dirty-bit by a dirty-bit based tracker may impact the correctness of these subsystems. For example, if the dirty bit is cleared for a disk-backed page by the dirty tracker at the beginning of the checkpoint interval without notifying the correct subsystem, the page write-back subsystem may never write the page to the disk. Second, the correctness of the dirty tracker can also be impacted by other subsystems. In the above example, if during a checkpoint interval, the page write back subsystem clears the dirty bit (after performing write-back to the disk for example), correctness of dirty tracking is also impacted. A similar and related challenge is when the virtual to physical mapping is changed because of features such as physical page migration and swapping. In short, access to dirty-bit from the tracker and other subsystems must be carefully (re)designed to ensure correctness.

In this paper, we propose LDT, a light-weight dirty tracking mechanism based on dirty-bit in PTE. Apart from the dirty-bit support from MMU, LDT assumes availability of two unused bits in the PTE to ensure correctness. LDT addresses the challenges mentioned above by maintaining extra meta-data in PTEs to ensure *API-level equivalence* for dirty-bit query and manipulation. For example, even if the dirty bit for a given PTE is cleared by LDT at the start of the interval, the dirty-bit checking API invoked by the rest of the subsystems in the OS will still return true for that PTE. The equivalence between a system with and without LDT is achieved by maintaining extra information in the PTEs to allow temporal state of the dirty-bit across checkpoint intervals. LDT also proposes a light-weight mechanism for making the dirty information available for user space consumption.

We design and implement LDT in the Linux OS (kernel version 5.5.10) in 64-bit Intel x86 system, and compare its performance against the state-of-the-art `SoftDirty` technique in the Linux OS. With write-heavy workloads, LDT outperforms `SoftDirty` by a factor of 2x in terms of the impact on the tracked application. For real-world workload benchmarks such as Redis [14], LDT provided an average ∼6% throughput improvement compared to the `SoftDirty` approach.

This paper makes the following contributions,

- Motivate a light-weight dirty tracking mechanism and present the challenges in designing memory dirty tracking solution using the dirty-bit support from the hardware.
- Design and implement LDT, a light-weight memory modification tracking mechanism in the Linux kernel for Intel x86-64 systems leveraging the hardware support for dirty-bits.
- Compare performance of existing write-fault based dirty tracking mechanism with LDT using a set of benchmarks.

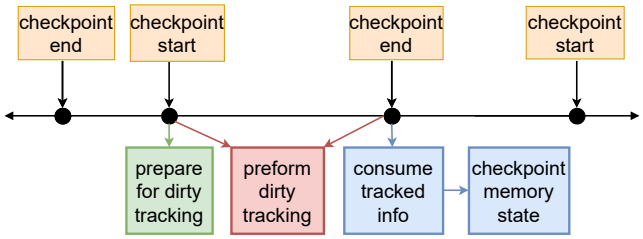


Fig. 1: Events during an incremental checkpoint of memory state of an application.

II. Background and Motivation

Typical applications using memory checkpointing use an iterative scheme where the initial checkpoint saves the complete memory state of the application(s)/VMs. In the subsequent iterations, the changes to the memory state are tracked such that the memory state of the application can be correctly rebuilt, when required. From the usage view, there are two entities involved in memory dirty tracking—the dirty tracker process (referred to as *tracker*) and the execution entity (process or VM) for which the memory dirty tracking is performed (referred to as *trackee*).

A. Iterative Memory Checkpointing

Figure 1 shows the tracker activities during an intermediate checkpoint interval for incremental dirty tracking. At the start of the interval (referred to as checkpoint start), the tracker performs preparation for tracking the dirty memory. Depending on the support for dirty tracking, the tracker may notify the system software such as the OS or hypervisor to mark the starting of the dirty tracking interval. After this point, the system software enables tracking memory modifications performed by the trackee. At the end of checkpoint interval, the tracker collects the dirty memory information from the system software using specified APIs. Depending on the usage scenario, the tracker may decide the start of the next checkpoint interval where the steps shown in Figure 1 are repeated.

A real-world implementation and usage of incremental checkpoint can be found in the Linux OS. Linux OS exposes the `SoftDirty` and `Pagemap` interfaces to perform memory dirty tracking of any process from the user space. One of popular users of the Linux OS incremental memory dirty tracking feature is Checkpoint/Restore In Userspace (CRIU) [15]. CRIU is used to checkpoint and restore any process and is extensively used for container migration and

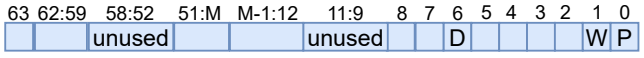


Fig. 2: Page Table Entry in Intel x86-64

cloning [5], [16]. When CRIU is used as the tracker, after capturing the initial memory state of a (trackee) process, CRIU uses the `SoftDirty` interface to notify the Linux kernel regarding start of a tracking interval for a given trackee process. CRIU collects the memory dirty information of the trackee process through the `Pagemap` interface at the end of checkpoint interval. To capture the memory state of the trackee in an incremental manner (e.g., during iterative migration of containers), CRIU may repeat the above steps a configurable number of times.

B. Page Dirty Tracking using Fault-on-write

Conventional systems provide mechanisms to track memory changes at page granularity using access restriction features of the memory management unit (MMU). Tracking dirty pages with a fault-on-write strategy works as follows. The system software can track changes to memory state at page-level granularity by forcing a page-fault on first write to any address within a page. One of the ways to achieve this is by write protecting the memory addresses by manipulating the access permission bits in the virtual to physical translation meta-data.

Figure 2 shows the page table entry (PTE) structure used to translate a 4KB page in the Intel x86-64 systems. Write access to an address in the range of a 4KB page is allowed only if the 'W' bit shown in the PTE structure is set to one. For dirty tracking of any given page, the OS can clear the write permission bit in the corresponding PTE, thereby causing a page fault on any write to the page [12]. The OS can note down the dirtied page address and allow subsequent writes to the page till the end of the interval. A process-level dirty tracking feature can be designed using the above strategy where at the start of the checkpoint interval for a given trackee process, the OS will remove write permission for the entire process address space. During the tracking interval, the OS can accumulate information regarding the dirtied pages and share it with the user space when required at the end of the tracking interval.

`SoftDirty` support in the Linux OS for x86-64 systems is an implementation of the fault-on-write mechanism. When requested by the user space (through the `procfs`), the Linux kernel clears the write permission bit (and the soft-dirty bit as explained below) in the PTEs of all mapped addresses by walking the page table of the trackee process. When the trackee process modifies any page for the first time during the tracking interval, the kernel handles the page fault and sets the soft-dirty bit in the corresponding PTE. The soft-dirty bit in PTE is one of unused bits as shown in Figure 2 which is cleared at the start of a dirty tracking interval. When the user space tracker wants to capture the dirty information, it can read the `pagemap` file present in the `procfs`. The read handler in the kernel walks through the page table of the process to collect the PTEs for the complete address space and formats them into the user space buffer before returning from the `read` system call. The `pagemap` feature is not tailor-made for dirty tracking and therefore returns the PTEs irrespective of their dirty status.

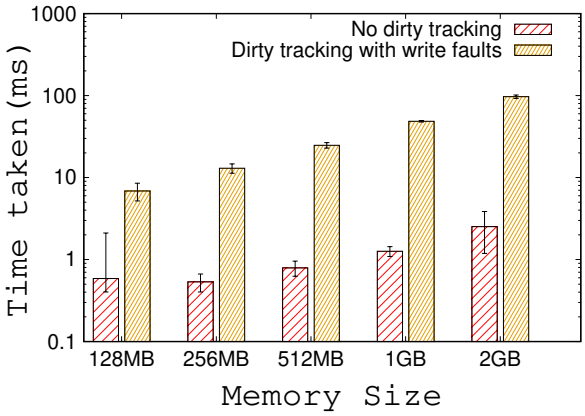


Fig. 3: Time taken (in milliseconds) to write 1 byte in each page of memory whose size is represented in X-axis in baseline case (no dirty tracking) vs write fault (dirty tracking) case. Y-axis is in log scale.

C. Motivation

Fault-on-write based approaches of dirty page tracking rely on the page-fault events generated by the hardware. This is required to provide the OS a point of intervention to enable tracking of the dirtied pages during a given interval. Introduction of page faults can have implications on the performance of the trackee. To study the extent of impact, we perform an experiment using the `SoftDirty` feature in an Intel i7 x86-64 system configured with the Linux OS. In this experiment, we execute a micro-benchmark which writes one byte to each page of the allocated memory of different sizes and captures the total time taken for all writes to complete. For dirty tracking, the benchmark invokes the `SoftDirty` interfaces after allocation and initialization of the memory area of the configured size.

Figure 3 shows the time taken to perform the write operations for two scenarios (dirty tracking enabled and disabled) with different sizes of allocated memory. When `SoftDirty` based dirty tracking is enabled, significant overheads can be observed. Compared to the case when dirty tracking is disabled, we observe that performance slowdown increases with increase in memory size. For example, the write time overheads with dirty tracking increases from ∼10x to more than 40x compared to the baseline (no dirty tracking) when memory size increases from 128MB to 2GB. The page fault overhead is non-trivial in case of `SoftDirty` which can impact the application performance during the checkpoint intervals. This observation clearly establishes the requirement of alternate solutions for efficient dirty tracking and motivates the design of LDT.

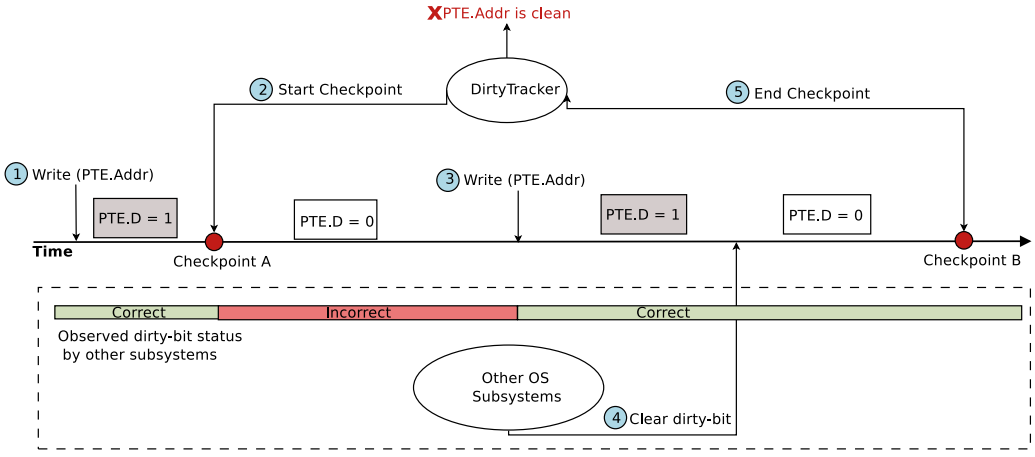


Fig. 4: Illustration of potential issues with dirty tracking solutions based on hardware dirty-bit support.

III. Design of LDT

A dirty tracking mechanism using the hardware support for PTE dirty-bits can meet the dirty-tracking requirements of the tools/programs at the upper layers in an efficient manner because dirty-bit based solutions do not incur overheads due to page-faults. However, there are some non-trivial challenges in designing such a solution. In the rest of the paper, we assume an underlying architecture with hardware support for dirty-bits.

A. Challenges

One of the primary challenges in design of LDT is ensuring correctness when dirty-bit is used simultaneously by many other subsystems in an OS. In an iterative dirty-tracking scenario (Figure 1), the OS is required to clear the dirty-bit at each checkpoint start and collect the dirty information at checkpoint end. This scheme works nicely when the dirty-bit in the PTE is manipulated only from the dirty-tracker. If other subsystems in the OS manipulate the dirty-bit out of turn, the correctness of dirty tracking is impacted. More seriously, manipulation of dirty-bit from the tracker may impact the correctness of other subsystems in the OS leading to system crashes. Therefore, design of LDT should address the above concerns, not only for correct dirty tracking, but also to ensure correctness of the entire OS.

Figure 4 highlights the integration issues involved in designing dirty tracking solution based on the hardware support for dirty bits where other OS subsystems consume and/or manipulate the PTE dirty bits. We show the actions on a single page (Figure 4 shows the corresponding PTE within the box representing the page) during one checkpoint interval between checkpoint A and checkpoint B. Consider that at ①, dirty-bit of the PTE is set by the hardware because the trackee process writes to an address within the page for the first time. When the dirty tracker initiates the checkpoint, dirty-bit for the page is cleared in step ②. Any other subsystem in the OS observes correct dirty status for the page till checkpoint A and gets the wrong dirty bit indicator afterwards. Consider in step ③ the process performs write to the page setting the dirty-bit again (from the hardware). Now, in step ④, some OS subsystem clears the dirty-bit from the PTE which results in collection of wrong dirty information by the dirty tracker for the page when the end checkpoint event (step ⑤) is triggered. In this case, the dirty tracker marks the page as clean (not dirty) even though the page is modified during the checkpoint interval (in ③). Note that, there are other interleavings resulting in either or both types of incorrectness.

B. Design Approach

The design of LDT makes following assumptions apart from the dirty-bit support from the MMU hardware. First, we assume at least two unused bits in the PTE for a valid mapping and one unused bit in the PTE of a swapped-out page. Most of the architectures leave some unused bits in the PTE for software use. For example x86, ARM, risc-v all have at least 2 bits reserved in the PTE for software use. Number of unused bits in the PTE for swapped-out page is dependent on the OS swapping implementation. Second, we assume all subsystems in the OS access/manipulate the dirty bit information in the PTE through a set of functions and do not access/manipulate the dirty bit from/in the PTE directly. This is a standard software engineering practice, especially for OSes written for variety of underlying architectures.

We use two unused bits (referred to as U_1 and U_2) in the following manner. U_1 is used to maintain a backup of the dirty information cleared by LDT such that any other OS subsystem checking the dirty-bit status can be served with the correct information. For the example scenario shown in Figure 4, any subsystem checking the status of the dirty bit for the PTE after checkpoint A, will be returned *true* because U_1 is used to maintain a backup of the dirty-bit status for the page while clearing the dirty-bit. U_2 is used to maintain a back-up dirty information for LDT in case some other OS subsystem clears the PTE dirty-bit. The erroneous conclusion by the dirty tracker can be corrected if U_2 is set when some other subsystem clears the dirty-bit (step ④ in Figure 4), and later used by the dirty tracker to interpret the page as dirty. Role of

the unused bit in the PTE for swapped out page (referred to as U_S) is to maintain the dirty information across swap-in and swap-out. If a page is already dirtied during a tracking interval and the OS wants to swap-out the page, the dirty information is maintained in the unused bit to avoid correctness issues as the structure of a PTE changes for swapped out pages.

LDT changes the dirty-bit query and manipulation interfaces such that it always provides a correct status to the rest of the OS. The dirty-bit status check for any PTE returns true if either the dirty-bit is set or U_1 is set. On the other hand, the dirty-bit clear for any PTE performs clears both dirty-bit and U_1, and sets the U_2 bit in the PTE. Similar manipulation of the APIs used for swapped-out PTEs are incorporated as part of LDT design. To exchange the page dirty information efficiently with the user space, LDT proposes a streamlined data exchange mechanism where the dirty information is provided in a compact manner.

C. Implementation Overview

We implement the proposed LDT scheme in the Linux kernel version 5.5.10 for Intel x86-64 systems by extending the `SoftDirty` framework. The implementation consists of around 200 lines of kernel code. `SoftDirty` provides user space interfaces to control the dirty tracking process as explained in Section II. We extended the interface to provide a new command to enable the user space to perform dirty-tracking using LDT. We also provide an alternate `procfs` API for the user space to collect page dirty information more efficiently.

We piggy-back the prototype implementation of LDT in on the existing `SoftDirty` implementation because of the following reasons. First, handling addition, removal or resizing of the address space areas (`vm_area` list in the PCB) is already implemented for the `SoftDirty` framework. Second, even when a process is not tracked, the kernel implementation of file mapped areas (with `MAP_SHARED`) for a process generate page faults on write which is used by the `SoftDirty` logic to track dirty pages. We do not want to disturb the file map functionality of the kernel. Third, `SoftDirty` already handles swapping and page migration using the designated bits in swapped PTEs. Reusing the same bit in the swapped PTEs, LDT avoids using additional bits in the swapped PTEs.

Set of high-level procedures presented in Algorithm 1 shows the working of LDT. The `RestartTrack` procedure in the kernel is invoked when any user space tracker initiates or restarts the tracking for a process (`app` in Algorithm 1). For every page mapped to a physical frame, if the dirty-bit is set, we remove the dirty bit and set the backup U_1 bit. We also remove the page dirty indicators such as $SoftDirty$ and U_2 bits to clear the residual history from previous tracking, if any. The $SoftDirty$ bit may be set in the previous interval because of the reasons explained above (for file mapped pages). If the page is swapped out, we clear the dirty-bit indicator from the swapped PTE (U_S in line#15). Finally, a full TLB flush is performed to remove stale entries from the TLB.

Algorithm 1: LDT implementation in the Linux kernel

```
   Input:
   app: Process to be tracked
   buf: User buffer for collecting page dirty information
   pte: Page table entry
 1 Function RestartTrack(app):
 2     Page P
 3     PTE pte
 4     foreach Page P in app.AddressSpace do
 5         pte = getpte(app, P)
 6         if (pte.Present) then
 7             if (pte.Dirty) then
 8                 pte.Dirty = 0
 9                 pte.U1 = 1
10             end
11             pte.SoftDirty = 0
12             pte.U2 = 0
13         end
14         if (pte.Swapped) then
15             pte.US = 0;
16         end
17     end
18     tlb_flush(app)
19     return
20 End Function
21 Function CollectDirty(app, buf):
22     bool Dirty = false
23     Page P
24     PTE pte
25     foreach Page P in app.AddressSpace do
26         Dirty = false;
27         pte = getpte(app, P)
28         if pte.Present then
29             if (pte.Dirty or pte.U2 or pte.SoftDirty) then
30                 Dirty = true
31             end
32         end
33         if pte.Swapped and pte.US then
34             Dirty = true;
35         end
36         if Dirty then
37             add_dirty_entry(P, buf)
38         end
39     end
40 End Function
41 Function CheckDirty(PTE pte):
42     return (pte.Dirty or pte.U1)
43 End Function
44 Function ClearDirty(PTE pte):
45     if (pte.Dirty) then
46         pte.U2 = 1
47     end
48     pte.Dirty = 0
49     pte.U1 = 0
50 End Function
```

During the tracking interval, any query to check the dirty status of the PTE of a page dirtied in the past using the modified `CheckDirty` API will return true until some kernel subsystem invokes the modified `ClearDirty` call. On a `ClearDirty` call, we clear both the dirty-bit and the U_1 bit, and set the U_2 bit in the PTE only if the dirty-bit is set (line #46 in Algorithm 1) because a set dirty-bit implies that the page is dirtied during the current tracking interval.

When the user space tracker collects the page dirty information, we walk through all the mapped pages and return the dirty page addresses back to the user space (using the buf argument in the `CollectDirty` procedure). A page is classified as dirty if at least one of the four indicators—the dirty-bit, the U_2 bit, the $SoftDirty$ bit and the U_S bit—is set. The page is also classified as dirty if the page belongs to a newly added

TABLE I: Micro-benchmark categories used in experiments

Benchmark	Description
Write-Only	Perform sequential writes to memory area
Read-Write	Perform fixed % of read and write to memory area
Write-Rate	Perform fixed number of writes per second to memory area

TABLE II: System Parameters

CPU	Intel i7-4770 CPU @ 3.40GHz
L1-D/I	32 KB (8 way)
L2	256 KB (8 way)
L3	8 MB (16 way)
DRAM	16 GB
OS	Ubuntu 18.04.3 LTS
Linux Kernel	5.5.10

virtual memory segment (not shown in Algorithm 1). There are many subtle integration issues, especially with swap and page migration logic, where the LDT dirty indicators are required to be transformed which is not shown in the Algorithm. Note that, the existing `pagemap` API in the Linux kernel collects dirty page information of all pages, irrespective of their dirty status. LDT `CollectDirty` procedure returns only the dirty pages saving the number of system calls and amount of data exchange between the user space and the kernel. The current prototype of LDT works only for 4KB pages.

D. Testing and Correctness

To validate LDT, we perform an extensive set of experiments with micro-benchmarks, stress tests, real applications and use cases. We create a set of micro-benchmarks to control the page modification and test the correctness of LDT by comparing the page dirty information against `SoftDirty`. Overnight tests with real-world applications such as Redis [14] with aggressive tracking (with one second tracking interval) is performed to make sure that LDT is not introducing any kernel issues (assert failures, crashes etc.). Further, the size of the Redis in-memory data store is pushed beyond the memory size to introduce swapping while LDT tracking is enabled to ensure correctness in extreme memory pressure scenarios. Finally, we perform iterative migration of an application container hosting Redis with a workload pushing the memory limits of the container. We observe that the Docker container is restored correctly and starts serving requests normally after restore.

IV. Evaluation

We compared the performance of LDT with no dirty tracking (`Baseline`) and conventional dirty tracking mechanism in Linux (`SoftDirty`). We evaluated LDT on a system with configuration mentioned in Table II using list of micro-benchmark categories mentioned in Table I.

A. Dirty tracking performance

LDT avoids initial write faults to dirty tracked pages and provides performance benefits for write intensive applications. We performed an experiment with different write footprints by changing the size of the memory mapped area of the application where every byte of the memory mapped region is written. Figure 5 shows the performance of dirty memory

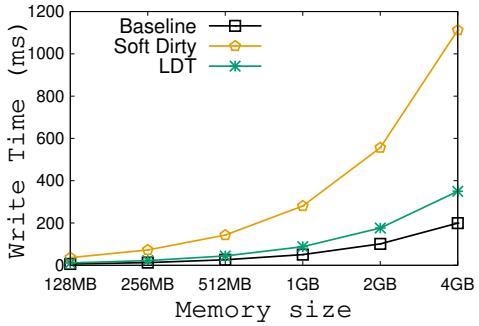


Fig. 5: Comparison of dirty tracking performance with write-only scenario. X-axis shows modified memory area size, Y-axis shows time taken to perform write.

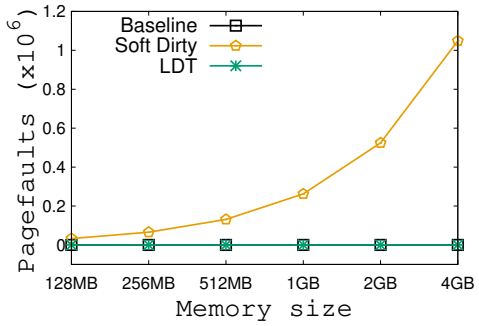
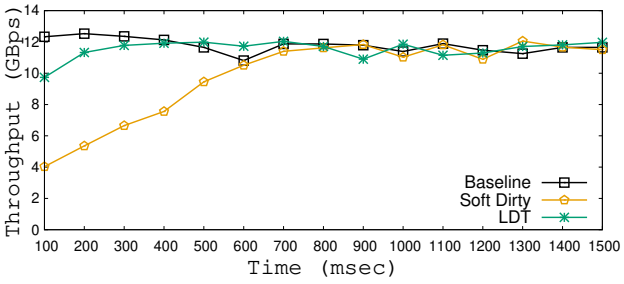


Fig. 6: Comparison of page-faults with different dirty tracking mechanisms in write-only scenario. X-axis shows modified memory area size, Y-axis shows number of page-faults.

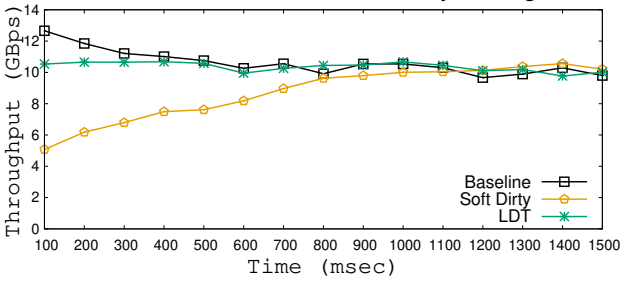
tracking with LDT under a write-heavy scenario by using write-only benchmark (Table I). We measured the time taken to perform writes with LDT and `SoftDirty` mechanisms where the baseline is with no dirty tracking. The benefit of using LDT in comparison with `SoftDirty` increases with increase in memory size. We observed more than 3x increase in time taken for writing 4GB memory with `SoftDirty` compared to LDT as shown in Figure 5. As shown in Figure 6, no additional page faults are introduced by LDT which leads to improved performance compared to `SoftDirty` where the first write to a tracked page results in a page fault.

As LDT avoids page fault on the first write to dirty tracked pages, its comparative benefit with `SoftDirty` increases with the number of writes during the tracking interval. We studied the application performance benefit of LDT with workloads consisting of different read to write ratios. Figure 7 shows the application throughput while being dirty tracked using LDT under different read-write percentages in a checkpoint interval. LDT resulted in similar throughput as of baseline in the initial phase of execution. The throughput gap between `SoftDirty` and LDT at the starting of a checkpoint interval widened with increase in write percentage. For example, we observe ~2.4x throughput in LDT compared to `SoftDirty` while performing 25% reads and 75% writes at the start of the tracking interval.

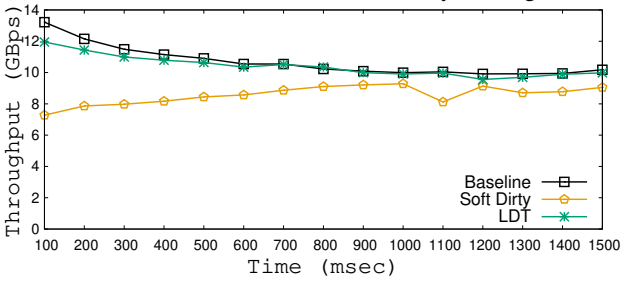
The `SoftDirty` throughput converges with baseline after-



(a) Performed 25% read and 75% write in dirty tracking interval.



(b) Performed 50% read and 50% write in dirty tracking interval.



(c) Performed 75% read and 25% write in dirty tracking interval.

Fig. 7: Comparison of LDT tracking performance with different read-write intensity. X-axis shows the time intervals in milliseconds, Y-axis shows application throughput with dirty tracking and no dirty tracking baseline.

wards because the page faults occur only on first write to dirty tracked pages. Subsequent writes from all three modes are equivalent; hence the performance is similar. Figure 7 clearly shows the benefit of LDT under all combinations of read-write percentages in an application where the benefits increase with more writes. The benefit of LDT can be attributed to avoidance of page-faults during the initial phases of the tracking interval as shown in Figure 8 . `SoftDirty` resulted in significantly higher number of page faults (Figure 8) at the starting of the checkpoint interval.

Applications may write to memory at different rates in a given checkpoint interval. The rate of writing decides the amount of page-faults in `SoftDirty` which will result in high CPU overheads near the start of checkpoint. Another parameter impacting the number of page faults is the writable working set size of an application. For a given rate of write, the number of page faults is expected to be higher for applications with larger working set sizes. To analyze the tracking performance of applications with different rates of write and writable

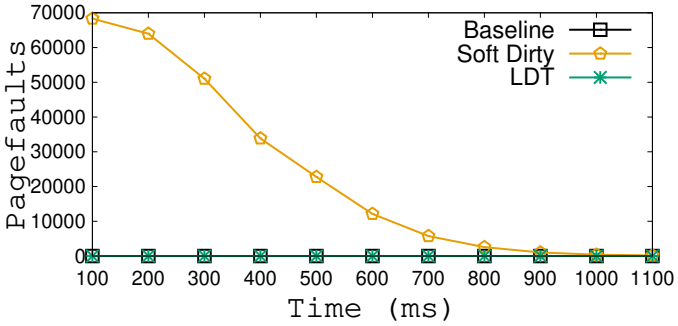
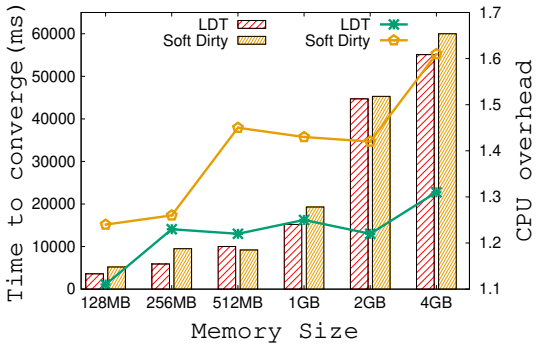


Fig. 8: Number of page-faults with 25% read and 75% write during the dirty tracking interval.

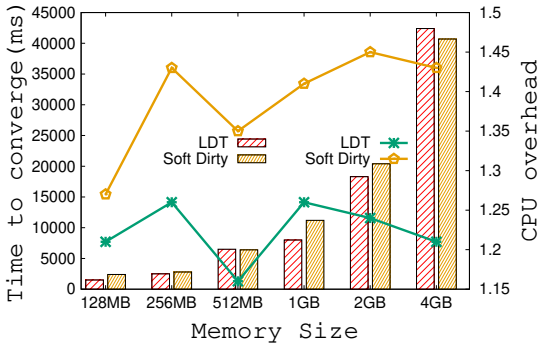
working set size, we performed the following experiment. We collected CPU utilization (in cycles every 100ms using the `perf` utility) for different combinations of writable working set size, write rate and dirty tracking technique using a parameterized micro-benchmark. `LDT` and `Soft Dirty` resulted in comparatively higher CPU utilization during the start of the experiment, and ultimately converged to the baseline CPU usage after some time.

Figure 9 shows the time taken to converge (Y1 axis, bar plots) with baseline CPU utilization while using `LDT` and `Soft Dirty`. We calculate the time of convergence for LDT and `SoftDirty` by using a difference threshold (from the median of the baseline) in the CPU utilization. For example, with write rate of 100K/sec and writable working set size of 1G, we calculate the median of the baseline CPU usage. We processed the CPU utilization samples of LDT for the same workload till we encounter a sample with CPU usage within a threshold of the calculated median value of the baseline to derive the time to convergence. The execution time while using dirty tracking converges quicker with increase in write frequency, i.e. with 128MB mapped memory area size, `SoftDirty` converged at 5200 μsec with 50K (Figure 9a), 2400 μsec with 100K (Figure 9b)and 600 μsec with 200K writes per second (Figure 9b). LDT performed better than `SoftDirty` for all write rates by converging earlier with the baseline execution time, thus attaining baseline application performance quicker.

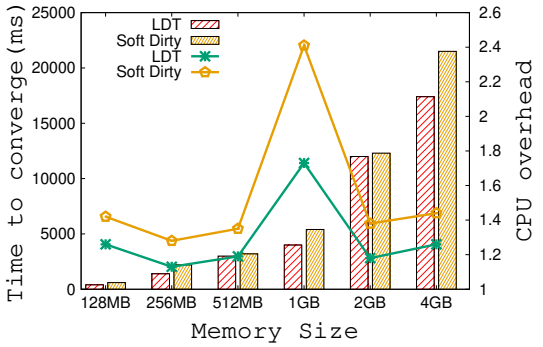
The CPU overhead on the Y2-axis (line plots) shows the ratio of total CPU cycles consumed till the time of convergence while using `LDT` or `Soft Dirty` to total CPU cycles consumed in the baseline setup till the same time. As shown in Figure 9, LDT outperformed `SoftDirty` mechanism as it resulted in comparatively lower CPU overhead. This benefit of LDT is from saving CPU cycles by avoiding write page fault service time. LDT resulted in an average $\sim$14% reduction in CPU overhead across all configurations with a maximum of 29% reduction in 200K write rate in case of 1GB mapped memory area. The execution time of an application converges with the baseline after the first write faults on the dirty tracked pages are over; thus the time taken to converge execution time depends upon the rate of write page-faults (i.e. rate of writes).



(a) Performed 50K writes per second.



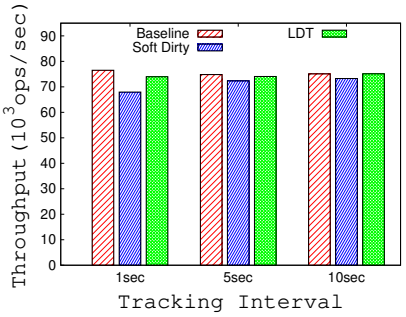
(b) Performed 100K writes per second.



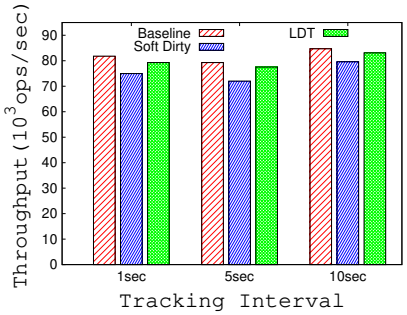
(c) Performed 200K writes per second.

Fig. 9: Comparison of LDT tracking performance under different write rates. X-axis shows mapped memory area size, Y1-axis shows time taken to converge with baseline (no dirty tracking) CPU usage. Y2-axis shows normalized CPU overhead (w.r.t. baseline) till the point of convergence.

We study the influence of dirty tracking on standard applications by tracking the in-memory database Redis [14]. We tracked Redis with LDT and `SoftDirty`, and performed *Get* and *Set* operations using YCSB [17] benchmark. We performed 90% *Set* (write/update) and 10% *Get* (read) as one configuration, and 50% *Set* and 50% *Get* in another configuration, to measure the influence of varying read-write percentage on Redis performance with different dirty tracking methods. Figure 10 shows that with LDT tracking, Redis throughput is closer to the baseline (no dirty tracking). LDT



(a) YCSB performed 50% read 50% write.



(b) YCSB performed 10% read 90% write.

Fig. 10: Comparative performance of dirty tracking with Redis benchmark for different tracking intervals. X-axis shows variation in tracking interval, Y-axis shows Redis throughput.

Read:Write (%)	Interval	Baseline (μs)	SoftDirty (μs)	LDT (μs)
10:90	1sec	77	84	79
	5sec	77	86	80
	10sec	74	80	74
50:50	1sec	79	92	82
	5sec	80	84	82
	10sec	80	84	81

TABLE III: 99^{th} percentile latency for write operations with Redis.

provides ∼2% to ∼8% throughput improvement as compared to `SoftDirty` . Maximum improvement of 8% is observed in 50% get and 50% set scenario with one second tracking interval.

Table III shows the 99^{th} percentile latency of write operations in Redis application for checkpoint intervals of 1, 5 and 10 seconds. For both the workload scenarios, we observe that the tail latency for `SoftDirty` is comparatively higher than `LDT`. This happens due to the time spent on handling page faults while servicing the write requests.

We also studied the overhead of dirty bit access and manipulation from other OS subsystems due to the introduction of extra reserved bits (U1 and U2) in case of LDT. We called the `checkDirty` and `clearDirty` methods (Algorithm 1) 5 million times from a kernel module to observe the overheads of LDT in comparison to the baseline Linux. We observed similar execution times in both the scenarios (LDT and unmodified Linux) which implies usage of extra bits in the PTE does not introduce any noticeable performance overheads.

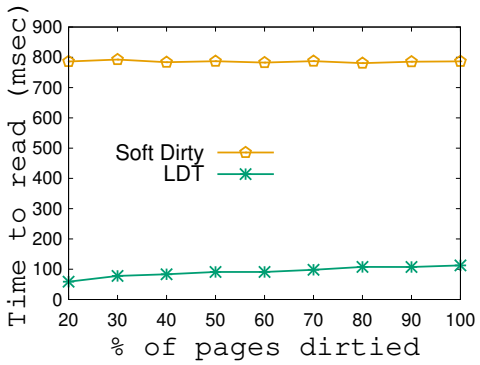


Fig. 11: Comparison of time to read dirtied page information using LDT interface and existing linux pagemap interface. X-axis shows percentage of pages dirtied in memory area, Y-axis shows time to read dirty tracked page information.

B. Dirty Page Information Collection

Applications need to consume the memory dirty information at the end of the checkpoint interval to copy the modified pages to a persistent store (Figure 1). With the conventional `SoftDirty` mechanism, applications can consume page dirty information by reading the `pagemap` interface in the Linux OS. `Pagemap` contains one 64-bit value for each virtual page. This 64 bit value contains information such as the physical address, a soft-dirty marker, swap status indicator etc. The application can offset into the `pagemap` file for pages of interest, and perform read operations to collect information regarding the soft-dirty status of pages. Using the `pagemap` interface for dirty tracking has the constant overhead of reading 64 bit values for all pages in a mapped area (in the worst case), irrespective of the actual number of pages dirtied in the interval. For example, an application with one GB mapped area has to read the information for all 2^{18} pages (i.e. number of pages in one GB mapped area) even if only one page is modified in the tracking interval. To address this shortcoming of `pagemap` interface, we expose a new interface in LDT which allows applications to collect information regarding pages dirtied during a checkpoint interval in the form of a list of dirtied pages. With the LDT specialized interface, information about only one page is returned instead of all 2^{18} pages in one GB space in the previous example.

We performed an experiment to study the performance benefits of LDT dirty page collection interface in comparison with `pagemap` interface. In this experiment, we dirty a certain percentage of pages of an one GB memory region and then observe the time taken to read the page dirty information using the `pagemap` interface and LDT interface. Figure 11 shows that the LDT interface significantly reduces time taken to read page dirty information in an interval. There is a consistent benefit (7x to 15x) with the LDT interface over `pagemap` interface, even for collecting information for high percentage of dirty pages (Figure 11).

Summary: LDT provides significant benefits in terms of reduced impact on the performance of the tracked applications, especially when the tracked applications are write-intensive. Even with moderate read scenarios, the tail latencies can be improved by LDT which can provide better QoS guarantees. LDT does not support tracking huge pages which we plan to incorporate as part of the future work.

V. Related Work

Incremental checkpointing Berkeley Labs Checkpoint Restart (BLCR) [1] is a checkpoint and restore mechanism for applications. It supports incremental checkpointing. Aurora Operating System [10] is a single level store that enables the persistence and manipulation of execution state. It uses CoW scheme to handle sharing of pages, objects. Speculative Memory Checkpointing (SMC) [18] is a technique that aims to achieve high frequency checkpointing by eagerly copying the hot pages while lazily tracing cold pages. Hardware assisted page modification tracking such as Intel PML [19] is limited to tracking memory modifications in virtualized systems. Lightweight Memory Check pointing (LMC), [20] a new user-level memory checkpointing technique that relies on compiler-based instrumentation to shadow the entire memory address space of the running program and incrementally checkpoint modified memory bytes in a LMC-maintained shadow state. Checkpointing solutions can use the light-weight dirty tracking support provided by LDT to realize efficient incremental checkpointing in a transparent manner.

Solutions leveraging MMU features: Usage of permission bits and unused bits in the PTE to realize different features and/or implement tools is not new. Apart from `Softdirty`, memory deduplication using Kernel same-page merging (KSM) [21], [22] uses a fault-on-write strategy. BadgerTrap [23] and DiME [24] modify the PTE access permissions to cause fault-on-access to build tools to capture TLB misses and emulate disaggregated memory, respectively. LDT only uses unused bits in the PTE without modifying access permissions.

Usage of incremental checkpointing: Checkpoint of application memory state has wide range of use cases such as software reliability [25], [26], debugging [27], container migration [28]–[30] and system consistency with non-volatile memory [31]. Most of the above applications use some form of fault-on-write mechanism to implement dirty tracking similar to the Linux `SoftDirty`. CRIU [32] freezes the processes at particular time, checkpoints them to persistent storage in an iterative manner, and then restores the processes from the point they were frozen using `SoftDirty`. Record/replay systems [33] [34] [35] record inputs to an application and replay the execution if required, such as for debugging purposes. These systems generate huge amounts of data unless incremental checkpointing is incorporated. Live migration of GPU [36] is an interesting use case of software page dirty tracking where GPU live migration can be achieved with low application down time. LDT provides an efficient alternative incremental checkpointing technique which can be used by many of the above usage scenarios.

VI. Conclusion

In this paper, we proposed LDT, a light-weight memory write monitoring mechanism to support efficient incremental checkpointing. Existing fault-on-write based approaches such as the Linux `SoftDirty` technique impacts the performance of monitored applications because of the additional overheads due to the introduction of page faults for the tracking purposes. LDT uses dirty bit and some unused bits in the PTE to identify the pages being written in a given interval. We implemented LDT in the Linux kernel for x86-64 architecture and performed correctness tests using real-world use cases such as container migration using CRIU. We evaluated the performance of LDT using a set of micro-benchmarks and real-world workloads such as Redis. Our evaluation thoroughly demonstrates the performance benefits of LDT compared to the state-of-the-art `SoftDirty` approach in the Linux OS. We observed performance improvement of write-intensive applications when monitored by LDT compared to `SoftDirty` monitoring. For example, LDT resulted in $\sim$2.4x improvement over `SoftDirty` for a workload with 75% write and 25% read operations. Our experiments with Redis key-value store when dirty tracking is enabled shows that LDT resulted in up to $\sim$8% throughput improvement over `SoftDirty`. LDT also provides a custom light-weight user interface to collect the memory dirty information in an efficient manner.

References

[1] M. Vasavada, F. Mueller, and P. H. Hargrove, "Comparing different approaches for incremental checkpointing : The showdown," in *Proceedings of OLS*, 2011.

[2] D. Vogt, C. Giuffrida, H. Bos, and A. S. Tanenbaum, "Lightweight memory checkpointing," *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pp. 474–484, 2015.

[3] B. Guler and Ö. Özkasap, "Efficient incremental checkpoint algorithm for primary-backup replication," *2017 25th Signal Processing and Communications Applications Conference (SIU)*, pp. 1–4, 2017.

[4] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live migration of virtual machines," in *Proceedings of the 2nd Conference on Symposium on Networked Systems Design and Implementation - Volume 2*, ser. NSDI'05. USA: USENIX Association, 2005, p. 273–286.

[5] C. Prakash, D. Mishra, P. Kulkarni, and U. Bellur, "Portkey: Hypervisor-assisted container migration in nested cloud environments," in *Proceedings of Virtual Execution Environments*, ser. VEE 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 3–17.

[6] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield, "Remus: High availability via asynchronous virtual machine replication," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI 08)*, 2008.

[7] S. M. Srinivasan, S. Kandula, C. R. Andrews, and Y. Zhou, "Flashback: A lightweight extension for rollback and deterministic replay for software debugging," in *Proceedings of the USENIX Annual Technical Conference*, 2004.

[8] E. Bugnion, V. Chipounov, and G. Candea, "Lightweight snapshots and system-level backtracking," in *HotOS*, 2013.

[9] Y.-M. Wang, Y. Huang, K.-P. Vo, P.-Y. Chung, and C. M. R. Kintala, "Checkpointing and its applications," *Twenty-Fifth International Symposium on Fault-Tolerant Computing. Digest of Papers*, pp. 22–31, 1995.

[10] E. Tsalapatis, R. Hancock, T. Barnes, and A. J. Mashtizadeh, "The aurora single level store operating system," in *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles*, ser. SOSP '21, 2021, p. 788–803.

[11] D. Bittman, P. Alvaro, P. Mehra, D. D. E. Long, and E. L. Miller, "Twizzler: a Data-Centric OS for Non-Volatile memory," in *Proceedings of the USENIX Annual Technical Conference*. USENIX Association, Jul. 2020, pp. 65–80.

[12] P. Guide, "Intel® 64 and ia-32 architectures software developer's manual," *Volume 3B: System programming Guide, Part*, vol. 2, no. 11, 2011.

[13] S. D. Bit, https://www.kernel.org/doc/html/v5.4/admin-guide/mm/soft-dirty.html.

[14] Redis, https://redis.io/.

[15] CRIU, https://en.wikipedia.org/wiki/CRIU.

[16] R. S. Venkatesh, T. Smejkal, D. S. Milojicic, and A. Gavrilovska, "Fast in-memory criu for docker containers," *Proceedings of the International Symposium on Memory Systems*, 2019.

[17] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM symposium on Cloud computing*, 2010, pp. 143–154.

[18] D. Vogt, A. Miraglia, G. Portokalidis, H. Bos, A. Tanenbaum, and C. Giuffrida, "Speculative memory checkpointing," in *Proceedings of the Middleware Conference*, ser. Middleware '15, 2015, p. 197–209.

[19] S. Bitchebe, D. Mvondo, L. Réveillère, N. de Palma, and A. Tchana, "Extending intel pml for hardware-assisted working set size estimation of vms," in *Proceedings of Virtual Execution Environments*, ser. VEE 2021, 2021, p. 111–124.

[20] D. Vogt, C. Giuffrida, H. Bos, and A. S. Tanenbaum, "Lightweight memory checkpointing," in *IEEE/IFIP International Conference on Dependable Systems and Networks*, 2015, pp. 474–484.

[21] ksm, https://www.kernel.org/doc/html/latest/admin-guide/mm/ksm.html.

[22] A. Garg, D. Mishra, and P. Kulkarni, "Catalyst: Gpu-assisted rapid memory deduplication in virtualization environments," in *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. New York, NY, USA: Association for Computing Machinery, 2017, p. 44–59.

[23] J. Gandhi, A. Basu, M. D. Hill, and M. M. Swift, "Badgertrap: A tool to instrument x86-64 tlb misses," *SIGARCH Computer Architecture News*, vol. 42, no. 2, p. 20–23, 2014.

[24] D. Buragohain, A. Ghogare, T. Patel, M. Vutukuru, and P. Kulkarni, "Dime: A performance emulator for disaggregated memory architectures," in *Proceedings of the 8th Asia-Pacific Workshop on Systems*, ser. APSys '17, 2017.

[25] G. Portokalidis and A. D. Keromytis, "Reassure: A self-contained mechanism for healing software using rescue points," in *International Workshop on Security*. Springer, 2011, pp. 16–32.

[26] J. F. Ruscio, M. A. Heffner, and S. Varadarajan, "Dejavu: Transparent user-level checkpointing, migration, and recovery for distributed systems," in *IEEE International Parallel and Distributed Processing Symposium*, 2007, pp. 1–10.

[27] D. Subhraveti and J. Nieh, "Record and transplay: partial checkpointing for replay debugging across heterogeneous systems," in *Proceedings of the ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, 2011, pp. 109–120.

[28] R. Stoyanov and M. J. Kollingbaum, "Efficient live migration of linux containers," in *International Conference on High Performance Computing*. Springer, 2018, pp. 184–193.

[29] S. Osman, D. Subhraveti, G. Su, and J. Nieh, "The design and implementation of zap: A system for migrating computing environments," *ACM SIGOPS Operating Systems Review*, vol. 36, no. SI, pp. 361–376, 2002.

[30] O. Laadan and S. E. Hallyn, "Linux-cr: Transparent application checkpoint-restart in linux," in *Linux Symposium*, vol. 159. Citeseer, 2010.

[31] N. Cohen, D. T. Aksun, H. Avni, and J. R. Larus, "Fine-grain checkpointing with in-cache-line logging," in *Proceedings of Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 441–454.

[32] CRIU, https://criu.org/Main_Page.

[33] Y. Chen, S. Zhang, Q. Guo, L. Li, R. Wu, and T. Chen, "Deterministic replay: A survey," *ACM Computing Survey*, vol. 48, no. 2, sep 2015.

[34] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen, "Revirt: Enabling intrusion analysis through virtual-machine logging and replay," *SIGOPS Operating System Review*, vol. 36, no. SI, p. 211–224, dec 2003.

[35] G. W. Dunlap, D. G. Lucchetti, M. A. Fetterman, and P. M. Chen, "Execution replay of multiprocessor virtual machines," in *Proceedings of Virtual Execution Environments*, ser. VEE '08, 2008, p. 121–130.

[36] J. Ma, X. Zheng, Y. Dong, W. Li, Z. Qi, B. He, and H. Guan, "Gmig: Efficient gpu live migration optimized by software dirty page for full virtualization," in *Proceedings of Virtual Execution Environments*, ser. VEE '18, 2018, p. 31–44.