Int J Parallel Prog (2015) 43:421–439
DOI 10.1007/s10766-013-0295-0

Efficiently Restoring Virtual Machines

Bernhard Egger · Erik Gustafsson · Changyeon Jo · Jeongseok Son

Received: 3 April 2013 / Accepted: 6 November 2013 / Published online: 22 November 2013

Abstract Saving the state of a running virtual machine (VM) for later restoration has become an indispensable tool to achieve balanced and energy-efficient usage of the underlying hardware in virtual desktop cloud environments (VDC). To free up resources, a remote user's VM is saved to external storage when the user disconnects and restored when the user reconnects to the VDC. Existing techniques are able to reduce the size of the checkpoint image by up to 80 % by excluding duplicated memory pages; however, those techniques suffer from a significantly increased restoration time which adversely affects the deployment of the technique in VDC environments. In this paper, we introduce a method to efficiently restore VMs from such space-optimized checkpoint images. With the presented method, a VM is available to the user before the entire memory contents of the VM have been restored. Using a combination of lazy-fetch and intercepting accesses to yet unrestored pages we are able to reduce the time-to-responsiveness (TTR) for restored VMs to a few seconds. Experiments with VMs with 4 GB of memory running a wide range of benchmarks show that the proposed technique, on average, reduces the TTR by 50 % compared to the Xen hypervisor. Compared to the previously fasted restoration of space-optimized checkpoints, the proposed technique achieves a threefold speedup on average.

B. Egger (✉) · E. Gustafsson · C. Jo · J. Son
School of Computer Science and Engineering,
Seoul National University, Seoul, Korea
e-mail: bernhard@csap.snu.ac.kr
URL: http://csap.snu.ac.kr/

E. Gustafsson
e-mail: erik@csap.snu.ac.kr

C. Jo
e-mail: changyeon@csap.snu.ac.kr

J. Son
e-mail: jeongseok@csap.snu.ac.kr

Keywords Virtualization · Checkpointing · Performance

1 Introduction

In recent years, virtualization has gained wide adoption in enterprises. Virtualization enables the consolidation of servers to reduce hardware cost, simplifies server management, and empowers enterprises to deploy massive virtual desktop infrastructures (VDI) [1] and virtual desktop clouds (VDC) [2]. In VDI/VDC environments, users connect remotely to their desktop running in a VM in the cloud [3]. While the contents of the virtual desktop persist across different sessions, the VM itself need not be active between sessions. To free up resources in the cloud, the VDC solution may choose to save the state of a user's VM to external storage and restore it when the user re-connects to his virtual desktop. With ever increasing amounts of memory allocated to VMs, the space-overhead of taking a snapshot (also called checkpoint) of a VM becomes more and more of a concern. The size of a running VM's snapshot is dominated by the amount of memory available to that VM [4]. Unless other measures are taken, a snapshot consumes at least as much space as it has memory. With typical memory sizes of 4 and more GBs, the space consumed by snapshots of inactive VMs becomes significant [5].

An even bigger problem is the time required to restore a VM. The snapshot of a VM is taken after the user terminates his session and thus goes unnoticed by the user. The time to restore a VM, however, is critical since the restoration process typically begins as soon as the user starts a new session. Assuming 8 GB of memory in the VM and the (unrealistic) single-user case where the full disk bandwidth of 200 MB/s is available, the restoration process from a snapshot would require more than 40 s before the VM becomes available to the user. Such a long time-to-responsiveness (TTR), i.e., the time until the VM is running and responsive [6], effectively prevents wide adoption of automatic snapshot/restoration for VDCs.

Several techniques have been proposed to address both the space- and time requirements of checkpointing and restoring a VM. Methods that aim at reducing the size of the snapshot [7–9] achieve a smaller checkpoint image at the expense of a longer restoration time or a reduced performance of the VM after restoration. Techniques that aim at fast restoration of VMs [6,10], on the other hand, achieve the short restoration time at the expense of increased disk space or a reduced performance of the VM.

In this work, we combine space-optimized snapshots with an efficient restoration technique. A space-optimized snapshot as proposed by Park et al. [8] excludes memory pages whose contents are duplicated on disk from the snapshot image. Most modern operating systems use some form of I/O buffering in main memory; hence the amount of duplication especially for long-running VMs is significant. Park et al. report a space-reduction of up to 80 % compared to standard snapshot images. The proposed technique restores a VM from a space-optimized snapshot by first loading the contents of the snapshot image into memory and then immediately restarting the VM. Duplicated data is read directly from external storage by a background process. In order to minimize the TTR a number of optimizations have been explored.

This paper makes the following contributions:

– We present a technique that reduces the restoration time of VMs from space-optimized snapshot images by delaying the loading of the entire memory contents of the VM. We identify and correctly handle all scenarios that could lead to a corruption of the VM's memory image during the delayed loading phase.
– We have implemented the proposed technique in the Xen VMM 4.1 environment and conduct a wide range of benchmarks with fully-virtualized VMs running Linux.
– We compare our work to original Xen, space-optimized snapshots [8], and the method presented by Zhang et al. [6]. In comparison with original Xen the proposed methods achieves a 50 % shorter TTR; compared to the restoring space-optimized snapshots, the presented method achieves a threefold speedup. Furthermore, the proposed method achieves a similar TTR to that of Zhang et al.'s approach [6] without any of the limitations of their method.

The rest of the paper is organized as follows. Section 2 discusses related research and gives a short introduction on virtualization and checkpoint/restore functionality. Section 3 describes the proposed technique in detail. The experimental setup and the results are discussed in Sects. 4 and 5. Finally, Sect. 6 concludes this paper and discusses future work.

2 Background and Related Work

Checkpointing and restoration techniques have been extensively studied in the operating systems community, typically in the context of failure recovery or fault tolerance in distributed systems. In that context, a number of researchers have tackled the problem of reducing the size of the checkpoints [11–13] or reducing the time to recover from a failure [14,15]. Plank et al. [11] propose compiler-assisted memory exclusion in which read-only or unused memory regions are excluded from incremental checkpoints. Heo [12] and Yi [13] optimize the disk space required for incremental checkpoints and the time to recover from them. Baker et al. [14] propose the use of a "recovery box" to store crucial process state needed for fast recovery. Li et al. [15] improve the recovery time from a process failure by tracking which pages are accessed immediately after taking a checkpoint. Upon a failure and the subsequent recovery, these pages are loaded first, and then the process is restarted.

In the context of virtual execution environments physical memory allocation to VMs and related functionality such as checkpointing, live migration, fault tolerance, and logging with replay have recently received a lot of attention in the research community. Reclaiming memory from a running VM without affecting its performance is not easy as the VMM has limited knowledge of a memory page's importance to the VM. A common technique is Ballooning [7] in which the VMM communicates with a driver running inside the VM. The ballooning driver requests non-shared memory that then cannot be used by the guest OS any more. The VMM can then reclaim this memory and allocate it to another VM. Most major VMMs, such as KVM [16], VMware [17], VirtualBox [9], and Xen [18], make use of the ballooning technique to reclaim memory from a running VM. The disadvantage of ballooning is that it requires a special driver running inside the VM which may be a problem in security-oriented setups. Additionally, the technique is built on the premise that the operating system

will page out unused pages or drop the page cache. Both measures lead to reduced performance of the VM due to increased I/O activity.

In VDI/VDC environments users connect to their desktop from a remote terminal using a protocol such as PCoIP. The desktop runs in a virtual machine. When users terminate the connection to the virtual desktop, the VM may be stopped and swapped out to disk to free up resources for other VMs. This process involves checkpointing the user's VM. As soon as the user reconnects to his desktop, the VM must be restored from the checkpoint.

A checkpoint of a running VM includes its disk, the volatile memory, the state of the (virtual) CPUs and all connected virtual devices. Typically, the disk of a VM does not need to be checkpointed since it is not shared between VMs or users. The checkpoint image thus comprises the VM's volatile state: its memory contents and the state of the VCPUs and all connected devices. To this day, all major VMMs [9,16–18] store a one-to-one memory image to disk. Even though some VMMs try to exclude unallocated pages or compress the data, the size of a checkpoint image is in the order of the VM's memory: a VM with 8 GB of RAM will produce an 8 GB checkpoint file. Accordingly, the time to create a checkpoint is dominated by the time required to write the memory data to disk.

2.1 Space-Optimized Snapshots

Researchers have proposed three techniques to reduce the size of a snapshot: (1) compress the checkpoint image, (2) exclude unused memory from the checkpoint image, and (3) exclude data that is available somewhere else, i.e., duplicated data, from the checkpoint image. The first technique, compressing the checkpoint image, is implemented in VirtualBox [9]. Compression has been found not to be very effective for checkpoint images [6]. The effectiveness of excluding unused memory depends on how much of memory the VM actually uses. Most OSes use unused memory as caches which reduces the amount of unused memory. This technique is also not easy to implement for fully-virtualized guests because the VMM has no a priori knowledge how the VM organizes its memory. Ballooning [7] tries to overcome these difficulties by allocating memory from the guest OS which can then be excluded from the snapshot.

Excluding duplicated data from checkpoint images has recently been proposed in prior work by Park et al. [8]. The technique is based on the observation that OSes cache disk blocks in memory to hide the long latency to external storage. This so-called *page cache* often occupies the better part of the available memory; page caches occupying 80 % of the available physical memory are common. The contents of the vast majority of blocks present in the page cache exist in identical form on disk; the exception are dirty blocks, i.e., write-cached blocks that are about to be written to disk. Park intercepts all I/O operations from the VM to external storage, and maintains an up-to-date mapping in the so-called `page map`. The `page map` contains an entry for each memory page in the VM's memory that is known to exist in identical form on external storage. To ensure consistency, duplicated pages must be mapped *read-only* in the VM's address space. A subsequent memory write operation to such a page then

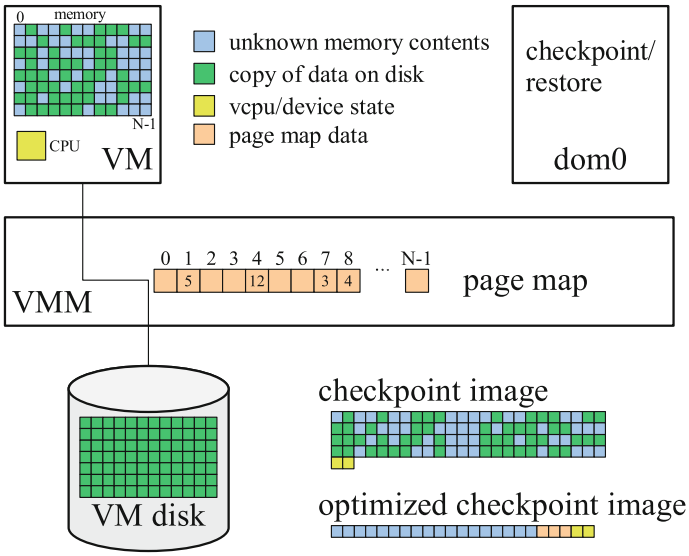

Fig. 1 Optimized checkpoint images

causes a page fault which is intercepted by the VMM and the mapping is removed from the `page map`. When a snapshot is taken, pages that have a known mapping to disk blocks, i.e., are present in the `page map`, are excluded from the optimized checkpoint image. Instead, the information of the `page map` is included in the snapshot image. Figure 1 illustrates the concept. Park reports no measurable slowdown by the I/O tracking. The method achieves a significant reduction in space and time for taking checkpoint images: on average, the optimized checkpoint image is reduced to 20 % of the size and saved in one third of the time compared to original Xen. The main disadvantage of Park's technique is that the time required to restore a VM from an optimized checkpoint image increases significantly. While a complete snapshot can be read with few I/O operations, restoring a VM from an optimized checkpoint requires loading the duplicated pages directly from the VM's external storage in possibly many, small I/O requests. The associated high seek overhead observed on mechanical (spinning) disks leads to a 50 % slowdown, on average, when restoring a VM with 4 GB of memory.

2.2 Optimized Restoration

Optimized restoration aims at reducing the time between the start of the restore and the time when the VM becomes available. The main idea here, lazy fetching, has been borrowed from live process/VM migration [19]. In a lazy-fetch scheme the restoration process starts the VM immediately, without restoring any or or only a small part of the memory. Whenever the VM tries to access memory that has not been restored yet, a page fault is triggered and the requested data is loaded from external storage. A number of optimizations to a pure lazy-fetch scheme have been proposed

to alleviate the slowdown experienced immediately after the VM has been restored. This slowdown is caused by the high number of page faults caused by accesses to yet unloaded pages. The most common technique in this direction is working set estimation [20].

Recently, Zhang et al. [6] have presented an implementation of lazy-fetch with working set estimation for VMWare checkpoint images. Since no working set information can be inferred from a VM's checkpoint, the working set has to be traced before/while the checkpoint image is saved to disk. The former, tracking accesses to pages before the checkpoint is taken, requires intercepting all memory operations and thus leads to reduced performance of the VM. For the latter, recording the working set while the snapshot is taken, the VM must be allowed to run for a short period of time after the checkpoint has been initiated. In their paper Zhang et al. use the latter method. The technique requires two copies of the snapshot image: one created when taking the checkpoint, and one when restoring it. This time and space overhead caused by creating copies of snapshots several GBs large renders this approach unsuitable for fast restoration in VDI/VDC environments.

Our technique improves the restoration process from optimized checkpoints [8]. The proposed technique enables VDI/VDC providers to restore VMs from space-optimized images within a few seconds. The next section describes the proposed method in detail.

3 Efficiently Restoring VMs from Checkpoints

The goal of efficiently restoring VMs is twofold: first, minimize the time until the VM becomes available to the user. Second, minimize the performance degradation of the VM after it has been restored.

The first goal can be achieved by starting the VM immediately after the VCPUs and the device states have been initialized. Memory accesses to pages that have not yet been loaded are intercepted and trigger restoring the page from the snapshot image. The almost immediate instantiation of the VM comes with a severe performance degradation caused by frequent page faults and the corresponding I/O activity. A common approach is thus to pre-load the working set of the VM before restarting it [6]. In addition to pre-loading the working set, we also propose a background thread which fetches yet unloaded pages into the VM's memory while the VM is running and several other optimizations to reduce the number of page faults. The following sections describe the design and implementation in detail.

3.1 Fast Restoration of a VM

The proposed technique is built on top of space-optimized checkpoints. The optimized checkpoints are similar to those proposed in [8] but extended to include additional information about the order in which duplicated pages were recognized. This information approximates the working set amongst the duplicated pages.

Restoring a VM from an extended optimized checkpoint progresses in the following steps:

1. *Instantiation* Instantiate the VM with VCPUs, devices, and memory pages from the data stored in the checkpoint image. While original Xen loads the entire contents of the memory from the snapshot image, memory pages containing duplicated data are not loaded but instead mapped *read-only*. We also create a background fetch process and initialize its lazy fetch queue (see below) with the list of pages that need to be loaded from external storage.
2. *Pre-loading* The pre-load phase loads the pages most recently added to the page map into the VM's memory *before* the VM is restarted. Pre-loading of pages is handled by the background process; the VMM simply waits for the process to finish loading of all pages designated to be in the pre-load set.
3. *Starting the VM* At this moment, the VM starts running and becomes accessible to the user. Note that the proposed method aims at minimizing the time-to-responsiveness: the TTR for a certain minimal utilization is typically only reached after the VM has been running for several seconds.
4. *Lazy Fetch* A background process in the VMM fetches the data of memory pages that have not yet been loaded from the VM's disk. For that purpose, the background fetch process maintains a *lazy fetch queue* which is initialized with the contents of the `page map` contained in the optimized checkpoint image. This is illustrated in Fig. 2 by the arrows labeled `(a)`. The lazy fetch queue consists of triplets containing the memory page number, denoted `PFN` (page frame number), the index of the (first) disk block, denoted `block number` (memory pages are typically bigger than one disk block, but written to consecutive disk blocks), and a `stale` flag (see below). Minimizing both the number of I/O requests and the disk seek overhead are key to achieve a short TTR. The lazy fetch queue is thus sorted in ascending order of the disk block number. A second index sorted by the `PFN` is maintained to allow efficient searches for memory pages. The background fetch process starts processing the elements in the lazy fetch queue immediately after the VM has been restarted. Communication with the VMM hypervisor is done asynchronously via

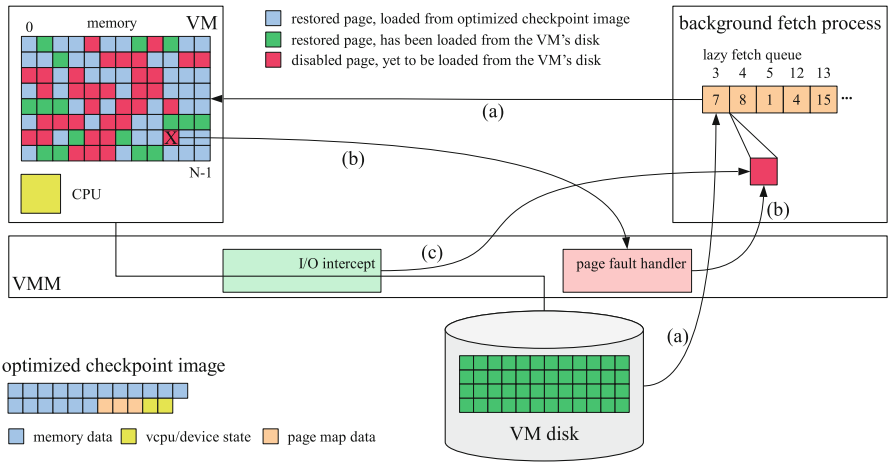

Fig. 2 Efficiently restoring VMs from an optimized checkpoint image

a ring buffer. The process priority of the background fetch process is lowered to give I/O requests of VMs higher priority.

5. *Page-fault Handling* Page faults triggered by the VM accessing a memory page that has not yet been restored are handled by the VMM's page fault handler. The requested page is loaded into the VM's memory, and the aborted instruction is restarted. The arrows labeled `(b)` in Fig. 2 illustrate this scenario.

The Xen hypervisor receives and dispatches all page faults that have been triggered by a VM violating the access permissions of a memory page. There are several possible sources of a page fault:

1. the guest OS inside the VM has deliberately mapped a memory page `invalid`, for example, to implement paging or catch null pointer dereferencing. Such page faults must be forwarded to the VM.
2. page faults caused by the paging mechanism of Xen. Such page faults are forwarded to and handled by `xenpaging`.
3. page faults triggered by an access to a lazily-fetched memory page that has not yet been loaded. These page faults need to be forwarded to the background fetch process.

The first two sources are already implemented and properly handled by the Xen hypervisor. The newly introduced third case can easily be distinguished from the other two causes by inspecting the PTE. The Xen hypervisor requests `xenpaging` to load the page which then notifies the background fetch process through a mailbox. The background fetch process checks the mailbox before loading the next element in the lazy fetch queue, i.e., handles page fault requests with high priority (Algorithm 1). As soon as the page has been loaded from external storage, the background process signals completion to `xenpaging` which then performs a hypercall to the hypervisor to resume the VM.

Page faults can occur while the data of a page is being loaded by the background fetch process. This can happen in any stage of the process of restoring a single page. The background fetch process restores the access permissions to the pages it has just loaded, and since the hypervisor and `xenpaging` communicate through a buffer, it is possible that the background fetch process in `xenpaging` receives a page fault for a page that has just been restored. `Xenpaging` checks the page's PTE before forwarding the request to the background fetch process. If the access permissions have already been restored, it silently ignores the request and resumes the VM. If the access permissions have not been restored yet, the request is send to the background fetch process which will simply discard the request and signal completion back to `xenpaging`.

3.2 Maintaining Consistency

Other than related work [6], the proposed method loads the memory contents directly from the VM's disk. Since the VM starts running before the entire memory has been restored, all I/O requests to disk need to be intercepted to ensure consistency. All I/O requests issued from a HVM guest pass through `ioemu`. We have modified `ioemu` to intercept and properly deal with each of the three possible causes of inconsistencies.

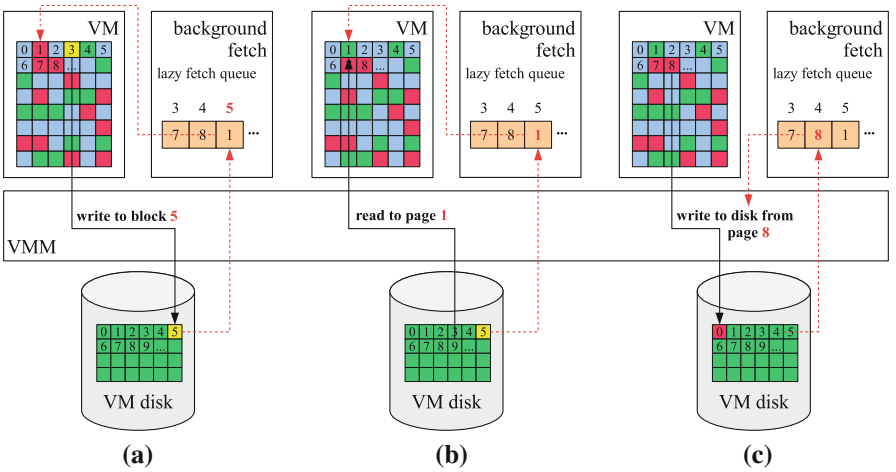

Fig. 3 The three race conditions that need to be handled to maintain consistency

Disk write race condition This situation occurs when the VM writes data to a disk block which has yet to be loaded by the background fetch process. If the write request is processed before the original data has been loaded, this will cause the background fetch process to read invalid data into memory. Figure 3a illustrates the case where the VM tries to write memory page 3 to disk block 5. Disk block 5, however, is still needed to restore the contents of memory page 1.

To detect a disk write race condition from a page m to a disk block b, `ioemu` searches the lazy fetch queue for an entry with disk block number b. If it finds such an entry, it sends a message requesting a high-priority load of block b to the background process. After loading the data into memory, the background process sends a message back to `ioemu` which then executes the write request initiated by the VM.

Disk read race condition The disk read race condition manifests itself when the VM reads data from disk into a page which has not yet been restored. This situation may occur if the guest VM discards the data of a memory page in the page cache and uses the page for another purpose. If the background fetch process later loads the data from disk, it will overwrite the newer data with old data which would lead to memory corruption. For disk read requests, the corresponding (and now stale) entry for the affected memory page can simply be discarded if the read request spans the entire memory page. For sub-page requests (for example, if the guest OS loads new data only into one half of the affected page), the affected page has to be loaded into memory before the read request can be executed. Figure 3b illustrates an example where the VM reads data from a disk block into memory page 1. Later, the background fetch process will load disk block 5 into memory page 1, thereby overwriting the new data from block 9 with the old data from disk block 5.

To detect a disk read race condition from a block b to a memory page m, `ioemu` searches the lazy fetch queue for an entry with the same memory page m. If such an entry exists, `ioemu` marks the entry as `stale` and continues with the read request. Since there is no explicit synchronization between `ioemu` and the background fetch process in this case, the latter checks the `stale` flag twice when processing an

element: once before loading the data from disk and once before copying the data into the VM's memory. If the `stale` flag is set before the data is loaded, the element is simply discarded. If the flag is set before copying the data, then `ioemu` may already have loaded the data from some other disk block and the copy operation is skipped. To prevent a race on the flag between reading the flag the second time and copying the data, the background process checks the flag and executes the copy operation inside a critical section.

Writing data to disk from a page that has not yet been loaded This race condition is less obvious and caused by the way VMMs virtualize the disk. Disk I/O requests are not sent directly to the (physical) disk, but go through a virtualization driver. In Xen 4.1, this driver runs in its own privileged driver domain. The guest OS' write request includes the address of the data to be written to disk in its own address space. This address is passed as a pointer, and because the data itself is not accessed so no page fault occurs. The privileged driver domain will eventually read the data from the page. However, since it has direct access to the VM's memory pages no page fault is raised and, consequently, corrupted data is written to disk.

To detect such writes, the I/O request is compared against the lazy fetch queue. If the affected memory page is still in the queue it is processed immediately. Then the I/O operation is resumed. Figure 3c illustrates the case where the VM writes memory page 8 to a disk block. Page 8, however, has not yet been loaded by the background fetch process. Since the VMM has unlimited access to the VM's memory, corrupt data is written to block 0 without I/O interception. In this last case, the VM requests to write data from memory page m to disk block b. `Ioemu` searches the lazy fetch queue for an element with PFN m, and if it detects such an element sends a request to the background fetch process. The background process loads the data from disk, copies it into the memory page m, and enables the PTE for the page. However, `ioemu` has already copied the data from the VM's memory into its own I/O buffers that now contain invalid data. The data of the requested page is thus sent along with the confirmation message back to `ioemu` which replaces the data into the I/O buffers before executing the write request. This is indicated by the dotted red arrow pointing back to the VMM in Fig. 3c.

Modern OSes perform the majority of the disk I/O on page granularity. I/O requests on a sub-page granularity are not tracked since doing so adds significant complexity but offers only a very modest reduction in the checkpoint image size. In the case of disk read races special care needs to be taken in the `ioemu` process to ensure correctness. The page request cannot be dropped by setting the `stale` flag, instead, `ioemu` requests a high-priority load of the page in question and waits for completion. Writes to yet unloaded pages are treated in a similar way.

3.3 Achieving a Short Time-to-Responsiveness

To achieve good performance after restarting a VM, it is important to minimize the number of page faults as well as the number of disk I/O operations caused by the background fetch process. The performance of the VM is much better when the lazy fetch queue is sorted by disk block number. This is somewhat counter-intuitive to

Algorithm 1 Background Fetch Process

```
Input: page map from snapshot image
  lfq ← page map
  while (not lfq.empty) do
    if (page fault pending) then
      block ← block which caused the page fault
    else if (high priority load pending) then
      block ← pending block
    else
      block ← next block in lfq
    end if

    blocks ← coalesce(block)
    read(blocks)

    for each block b in blocks do
      if (not b.stale) then
        copy data to VM memory
      end if
      remove b from lfq
    end for
  end while
```

the principle of locality according to which the lazy fetch queue should be sorted by PFNs, the memory page numbers. The reason why the former performs better is that it (a) reduces the disk seek overhead and (b) allows coalescing and loading of consecutive entries in one I/O request. The maximum number of pages coalesced into one request is a design parameter: if we allow only a few pages to be loaded by one I/O request, the number of I/O requests increases. If too many pages are bundled into one request, then fetching the data will take more time which will cause the TTR to drop. This trade-off is analyzed in the result section (Sect. 5). Coalescing is implemented with optional *hole support*: entries that are not consecutive but separated by only a few blocks ("holes") are coalesced in order to minimize the number of page faults (Algorithm 2).

The disk optimizations used in this paper aim at spinning disks since those are still the most commonly found non-volatile storage medium in computers today. Solid-state drives (SSDs) may require different optimizations since they incur no seek overhead.

3.4 Implementation

Algorithms 1 and 2 illustrate the background fetch process and the coalescing in pseudo-code form. The background fetch process (Algorithm 1) is created during the instantiation step (see Sect. 3.1). The *lazy fetch queue*, lfq, is a doubly-linked list sorted in ascending disk block order and initialized with the `page map` from the snapshot image. Each element in the list represents a duplicated memory page whose contents still have to be restored, i.e., the background fetch process terminates as soon as the lazy fetch queue is empty. Page faults and other high-priority requests are sent to the background process and handled first. To achieve a better overall responsiveness,

Algorithm 2 Coalesce

```
Input: block block to load, cf coalescing factor
  max_blocks ← 0
  max_set ← ∅
  for start ← block − cf − 1 to block + cf − 1 do
    num_blocks ← 0
    set ← ∅
    for b ← start to start + cf do
      if (b ∈ lfq and not b.stale) then
        num_blocks ← num_blocks + 1
        set ← set + {b}
      end if
    end for
    if (num_blocks > max_blocks) then
      max_blocks ← num_blocks
      max_set ← set
    end if
  end for
  return max_set
```

outstanding load operations of blocks around the selected *block* are coalesced (see below), then read into a memory buffer. Each loaded block's *stale* flag is then checked again (see Sect. 3.2) before the data is copied into the VM's memory and the page is enabled.

The implementation of the `coalescing` method is outlined in Algorithm 2. The method checks each span of cf, the maximum coalescing factor, blocks around the selected block and then selects the span with the most outstanding blocks. The actual implementation is optimized to use a single `for`-loop and a ring buffer with a runtime of $O(cf)$ instead of the nested `for`-loops.

4 Experimental Setup

We have implemented the proposed technique in the Xen hypervisor version 4.1.2 [21] on a machine with an Intel(R) Core(TM) i5-2500 CPU @ 3.30 GHz, 16 GB RAM, and a Western Digital HDD with a maximal read throughput of 170 MB/s. Dom0 runs Ubuntu Server 12.04.1 LTS. The guests have been configured to run Ubuntu 10.04 with 4 GB of RAM and 2 VCPUs.

We have executed a number of different benchmarks representing various usage patterns that have used in related work. Table 1 lists the benchmarks. The user sessions benchmarks include the `movie` and `desktop` benchmark. For the `movie` benchmark, a movie is played. The `desktop` benchmark contains active office and Internet programs controlled by automation scripts. `Gzip` compresses a large file; this benchmark is both I/O and CPU-intensive. `Make` compiles the Linux 2.6.32.60 kernel and represents a CPU-intensive benchmark. Copying a file and running Postmark are I/O-intensive loads (`copy` and `postmark`).

To create the snapshots the VM for each benchmark is booted up and idles for some time, then the actual benchmark is started inside the VM. The snapshot is taken before the benchmark finishes in order to measure the benchmark performance across a

Table 1 Benchmark scenarios and snapshot size

Benchmark	Description	Size of snapshot (%)
`movie`	Playing a movie (file size: 700MB) using mplayer	23
`desktop`	Four Libre Office Writer documents of about 10 MB each, Firefox with four open tabs	23
`gzip`	Compressing a file of 512 MB random data using gzip	51
`make`	Compiling the Linux 2.6.32.60 kernel (default configuration)	32
`copy`	Backing up a file of 1 GB in size	17
`postmark`	File system benchmark (128 files, 400 transactions, file size 4K-10M)	40

checkpoint/resume cycle. The third column in Table 1 displays the size of the optimized snapshot compared to that of unmodified Xen (4 GB).

All experiments are performed on fully-virtualized (HVM) guests. The proposed technique can also be applied to para-virtualized (PV) guests; in fact, PV guests will achieve better results thanks to the possible interaction between the hypervisor and the guest OS. For example, currently unused memory pages have to be saved and restored for HVM guests because no assumptions can be made about the memory management of the guest OS. For PV guests, unused memory pages can be easily excluded and need not be restored.

5 Experimental Results

The proposed technique aims at reducing the restoration time while keeping the performance degradation of the VM to a minimum. The two measures of interest are thus (1) the time until the VM is restarted and (2) the performance degradation introduced by the proposed lazy fetch technique. Zhang proposes the $TTR(w, u)$, the *time-to-responsiveness* measure [6], which denotes the point after which a VM achieves a certain minimal utilization u within a given time-window w. We compare the proposed method to the unmodified Xen hypervisor, the method proposed by Park [8], denoted `orig. pagecache`, and Zhang's technique.

5.1 Restoration Time

The restoration time denotes the duration from initiating the restoration of a checkpoint until the VM starts running. For unmodified Xen and the proposed method, the time to restore a VM from a snapshot is proportional to the size of the checkpoint image. Unmodified Xen always loads 4 GB of data independent of the benchmark; with our method the amount of data depends on the amount of duplication in the VM.

Fig. 4 Normalized restoration time without working set pre-loading

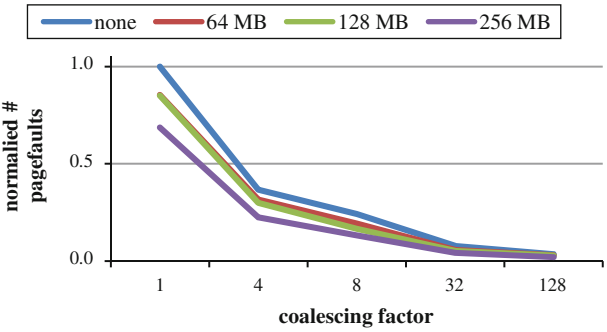

Fig. 5 The effect of pre-loading and coalescing on the number of pagefaults

`Orig.pagecache` loads the duplicated pages from the VM's disk image and thus incurs a significant overhead. Figure 4 shows the results for `orig.pagecache` and the proposed method with `none`, `64 MB`, `128 MB`, and `256 MB` of working-set pre-loading. Even though we have optimized the restoration method in Park's approach, `orig. pagecache`, reading the duplicated data from external storage results in an average slowdown of around 40 %. The proposed method requires, on average, only 40 % of the time of unmodified Xen to restore a VM if no working-set preloading is performed.

Figure 5 shows the effect of pre-loading 64, 128, or 256 MB of the working-set on the number of page faults in dependence of the coalescing factor averaged over all benchmarks. If no coalescing is performed, i.e., only the next page in the lazy-fetch queue or the page that triggered a pagefault is loaded, then preloading 256 MB of data results in a 30 % reduction of page faults. We observe that the coalescing factor has a much more significant effect on the number of pagefaults than working-set preloading. For a coalescing factor of 32 without preloading, for example, the number of pagefaults is reduced to 7 %, and preloading 256 MB of data reduces this number only by an additional 2 %. The reason for the poor performance of preloading can be explained by the way the working set is defined: since we only track the promotion of disk blocks into memory but not subsequent memory operations to the corresponding page, the accuracy of the working set is rather poor. Maintaining an accurate working

set would require tracking memory read accesses to duplicated pages; this would incur an unacceptably high overhead.

The increase in restoration time caused by pre-loading by far outweighs its benefits. Therefore, in the remaining experiments no WS pre-loading is performed.

5.2 Time-to-Responsiveness

While unmodified Xen and previous work restore the contents of the VM in its entirety, the proposed technique restarts the VM before pages duplicated on disk have been loaded. A background process is loading these pages into the VM's memory while the VM is running; nevertheless, the VM may try to access a page that has not yet been restored. The page fault triggered by this access causes the VMM to stop the VM, handle the page fault by loading the requested page, before the VM and the aborted instruction can be restarted.

To measure the performance degradation of the restored VMs, we adopt the same approach as Zhang [6]: even though the VM is restored and running after a few seconds, frequent page faults interrupt the VM and reduce the performance to the point where the VM appears unresponsive to the user. Our goal is thus to measure the point in time from which on forward the VM always achieves a certain minimum utilization. This measure is represented by the *time-to-responsiveness* and is defined as

$$TTR(w,u) = min\left\{t | MMU_{\forall t' \geq t}(w) \geq u\right\}$$

where $MMU(w)$, the *minimum mutator utilization*, denotes the minimal level of utilization (CPU time) the mutator (the VM) achieves in a given time window w. A $TTR(1.0\,\text{s}, 80\,\%) = 5\,\text{s}$, for example, states that after five seconds the VM always achieves at least 80 % utilization within any 1-s window. The time window w and the desired utilization u are application-specific parameters. Similar to Zhang [6], we set $w = 1.0\,\text{s}$ and $u = 80\,\%$ for interactive benchmarks (`movie`, `desktop`) and $u = 50\,\%$ for the non-interactive benchmarks (`gzip`, `make`, `copy`, and `postmark`).

Handling a page fault stops the VM and thus reduces the level of utilization available to the VM. To achieve a short TTR, it is imperative to reduce the number of pagefaults at much as possible, and to trigger page faults early or avoid them all together. As shown in Fig. 5, coalescing is very successful at reducing the pagefaults. Since the VMM has no knowledge of which pages the VM is going to access, triggering page faults early is impossible. Instead, we can try to avoid page faults by fetching the pages that have not yet been loaded from external storage by a background process. Figure 6 shows the effect of the different optimizations on the number of pagefaults for (a) `desktop`, a representative of an interactive benchmarks, and (b) `postmark`. The other benchmarks exhibit a similar behavior.

In `naïve lazy` only 32-page coalescing is enabled. `Hole support` supports coalescing even if not all pages are consecutive. `Backup support` enables coalescing in both directions of the page fault and finds the optimal window of 32 pages that still includes the page fault but results in the most pages loaded. `Background fetch` shows the effect of fetching pages in the background using the lazy-fetch

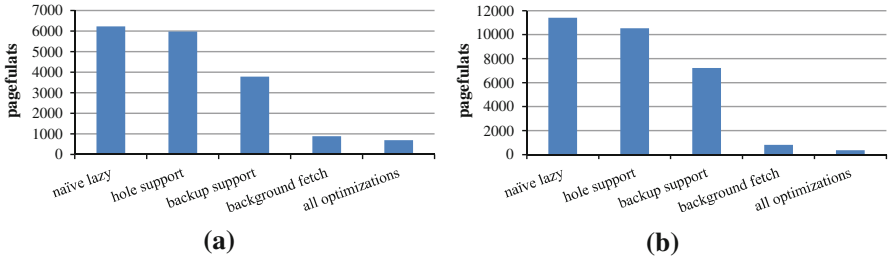

Fig. 6 The effect of optimizations on the number of page faults. **a** desktop, **b** postmark

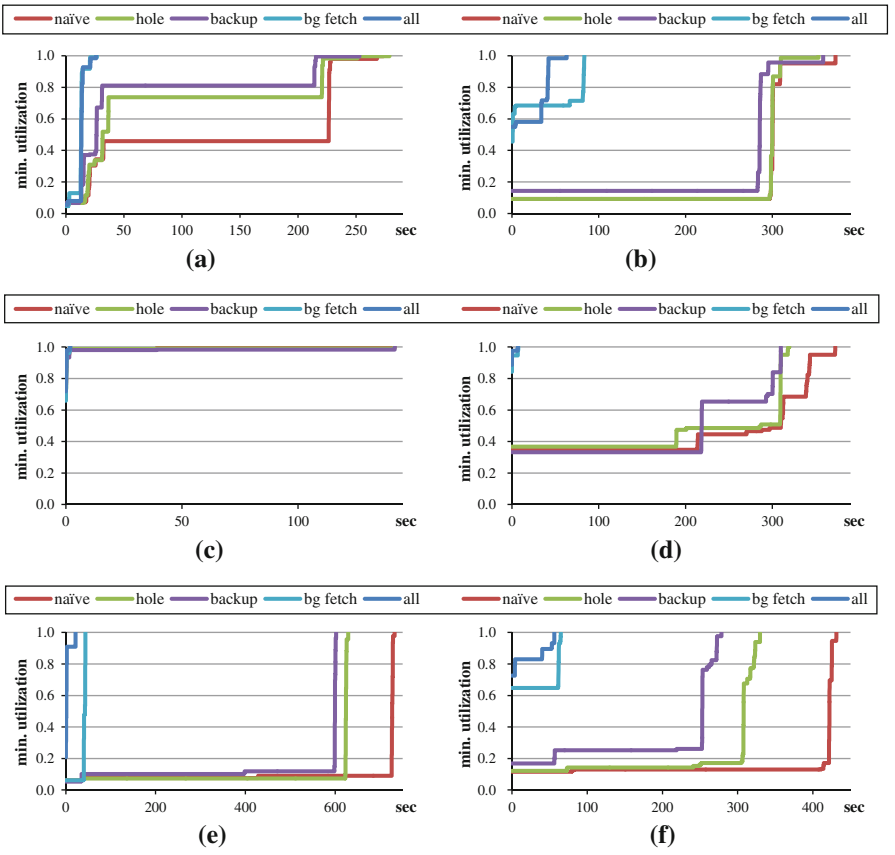

Fig. 7 The effect of optimizations on the time-to-responsiveness. **a** desktop, **b** postmark, **c** movie, **d** gzip, **e** make, **f** copy

queue. Finally, `all optimizations` shows the results when all optimizations are enabled. The results clearly show the effectiveness of the background fetch process. This is not obvious: the duplicated data mostly stems from the guest OS'es page cache, and it is not a priory clear whether reloading the page cache is more beneficial than simply dropping the page cache and fetch the pages again.

Fig. 8 Restoration time and TTR

In Fig. 7 the effect of the different optimizations on the minimal mutator utilization and the TTR is shown. Since the TTR measures the latest point in time after which a certain minimal utilization can be guaranteed, a series of page faults long after the VM has been started will result in a high TTR value. Is is thus not surprising that the background fetch queue, `bg fetch` is the optimization that has the biggest positive effect on the TTR value as it pre-fetches pages in the background and thus effectively reduces the number of page faults if the page is loaded before the VM accesses it. The TTR graphs also show that it is clearly beneficial to load the page cache instead of simply purging it (which is what the ballooning technique basically does) if the goal is to achieve a guaranteed minimal utilization as soon as possible.

Figure 8, finally, shows the breakdown of the restoration time (the time until the VM becomes accessible) and the time until the VM reaches the designated TTR. For most benchmarks, the TTR is very short. In the case of `desktop` the applications inside the VM trigger a number of pagefaults relatively late after the VM has been restarted which results in a long TTR.

Overall, the proposed method achieves very fast restoration times from checkpoint images that are significantly smaller than those of related work. Compared to unmodified Xen, on average, the VM achieves its "usable" state about 50 % sooner, i.e. after about 12 s. Zhang `et al.` report TTR values between 20 and 80 s for VMs with 1 GB of memory; our method clearly outperforms their approach.

6 Conclusion

We have proposed a technique for efficiently restoring VMs from snapshots that do not include data for memory pages whose contents are duplicated on external storage. Instead of loading the entire memory of the VM from disk before resuming the VM, the VM is restarted after only partially restoring its memory contents. Pages that still need to be loaded from disk before they can be accessed are protected by marking them invalid in the VM's page tables. A background process fetches pages into the VM's memory; and accesses to pages that have not yet been restored are intercepted by a page fault handler. Several optimizations such as coalescing accesses to neighboring

disk blocks and pre-loading a a varying number of pages significantly improve the performance of our solution.

The proposed technique has been implemented in the Xen 4.1 hypervisor and compared against original Xen and related work. Compared to Xen, the VM reaches the required minimal utilization after only 50 % of the time for optimized checkpoint images that require 40 % of the space. Compared to related work, we show that the proposed technique achieves significantly improved time-to-responsiveness. On average, a VM with 4 GB of memory achieves the minimally required utilization after 12 seconds; a value we believe enables the use of the proposed technology in VDI/VDC environments.

Acknowledgments We thank the reviewers for the helpful and constructive feedback. This research was supported by the Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT & Future Planning (2012R1A1A1042938). ICT at Seoul National University provided research facilities for this study.

References

1. Miller, K., Pegah, M.: Virtualization: virtually at the desktop. In: Proceedings of the 35th annual ACM SIGUCCS fall conference. ACM, SIGUCCS'07, New York, NY, USA, pp. 255–260 (2007)
2. Sridharan, M., Calyam, P., Venkataraman, A., Berryman, A.: Defragmentation of resources in virtual desktop clouds for cost-aware utility-optimal allocation. In: Utility and Cloud Computing (UCC), 2011 Fourth IEEE International Conference on, pp. 253–260 (2011)
3. Citrix: Xen Desktop 7. http://www.citrix.com/products/xendesktop/ (2013)
4. Laadan, O., Nieh, J.: Transparent checkpoint-restart of multiple processes on commodity operating systems. In: 2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference. USENIX Association, ATC'07, Berkeley, CA, USA, pp. 25:1–25:14 (2007)
5. Sancho, J.C., Petrini, F., Johnson, G., Fernandez, J., Frachtenberg, E.: On the feasibility of incremental checkpointing for scientific computing. Int. Parallel Distrib. Process. Symp. **1**, 58b (2004)
6. Zhang, I., Garthwaite, A., Baskakov, Y., Barr, K.C.: Fast restore of checkpointed memory using working set estimation. In: Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments. ACM, VEE '11, New York, NY, USA, pp. 87–98 (2011)
7. Waldspurger, C.A.: Memory resource management in vmware esx server. SIGOPS Oper. Syst. Rev. **36**, 181–194 (2002)
8. Park, E., Egger, B., Lee, J.: Fast and space-efficient virtual machine checkpointing. In: Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments. ACM, VEE'11, New York, NY, USA, pp. 75–86 (2011)
9. VirtualBox. http://www.virtualbox.org (2013)
10. Koto, A., Yamada, H., Ohmura, K., Kono, K.: Towards unobtrusive vm live migration for cloud computing platforms. In: Proceedings of the Third ACM SIGOPS Asia-Pacific conference on Systems. APSys'12, Berkeley, CA, USA, USENIX Association, pp. 7–7 (2012)
11. Plank, J.S., Beck, M., Kingsley, G.: Compiler-assisted memory exclusion for fast checkpointing. IEEE Tech. Comm. Oper. Syst. Appl. Environ. **7**, 10–14 (1995)
12. Heo, J., Yi, S., Cho, Y., Hong, J., Shin, S.Y.: Space-efficient page-level incremental checkpointing. In: Proceedings of the: ACM Symposium on Applied computing. ACM, SAC '05, New York, NY, USA, pp. 1558–1562 (2005)
13. Yi, S., Heo, J., Cho, Y., Hong, J.: Adaptive page-level incremental checkpointing based on expected recovery time. In: Proceedings of the: ACM Symposium on Applied computing. ACM, SAC '06, New York, NY, USA, pp. 1472–1476 (2006)
14. Baker, M., Sullivan, M.: The recovery box: Using fast recovery to provide high availability in the unix environment. In: Proceedings USENIX Summer Conference, pp. 31–43 (1992)
15. Li, Y., Lan, Z.: A fast restart mechanism for checkpoint, recovery protocols in networked environments. In: Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on, pp. 217–226 (2008)

16. Habib, I.: Virtualization with kvm. Linux J. **2008** (2008)
17. VMware Workstation. http://www.vmware.com/products/workstation (2013)
18. Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., Warfield, A.: Xen and the art of virtualization. In: SOSP '03: Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles. ACM, New York, NY, USA, pp. 164–177 (2003)
19. Hines, M.R., Gopalan, K.: Post-copy based live virtual machine migration using adaptive pre-paging and dynamic self-ballooning. In: VEE '09: Proceedings of the: ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments. ACM, New York, NY, USA, pp. 51–60 (2009)
20. Jiang, S., Chen, F., Zhang, X.: Clock-pro: an effective improvement of the clock replacement. In: Proceedings of the Annual Conference on USENIX Annual Technical Conference. ATEC '05, Berkeley, CA, USA, USENIX Association, pp. 35–35 (2005)
21. The Xen Hypervisor. http://www.xen.org (2013)