Zeroing memory deallocator to reduce checkpoint sizes in virtualized HPC environments

Journal of Supercomputing manuscript No.
(will be inserted by the editor)

Zeroing Memory Deallocator to Reduce Checkpoint Sizes in Virtualized HPC Environments

Ramy Gad · Simon Pickartz · Tim Süß · Lars Nagel · Stefan Lankes · Antonello Monti · André Brinkmann

Received: date / Accepted: date

Abstract Virtualization has become an indispensable tool in data centers and Cloud environments to flexibly assign virtual machines (VMs) to resources. Virtualization also becomes more and more attractive for High-performance Computing (HPC). This is mainly due to the strong isolation of VMs which enables: (1) the sharing of cluster nodes and optimization of the system's overall utilization; (2) load balancing by means of migrations due to the reduction of residual dependencies; and (3) the creation of system-level checkpoints increasing the fault tolerance in an application-transparent way.

On the downside, the additional virtualization layer conceals information that is only available on the process level. This information has a direct influence on the checkpoint size which should be kept as small as possible. In this paper, we propose a novel technique for checkpoint size reduction in virtualized environments. We exploit the fact that the hypervisor detects zero pages which are omitted when capturing a checkpoint. Moreover, compression techniques are applied for a further reduction of the checkpoint size. We therefore fill freed memory regions with zeros supporting both the zero page detection and the compression. We evaluate our approach by taking the example of HPC applications. The results reveal a reduction of the checkpoint size by up to 9 % when compression is disabled in the hypervisor and up to 49 % with compression enabled. Furthermore, memory zeroing is able to reduce VM migration time by up to 10 % when compression is disabled and by up to 60 % when compression is enabled.

Ramy Gad, Tim Süß, and André Brinkmann
Zentrum für Datenverarbeitung, Johannes Gutenberg-Universität Mainz, E-mail: {gad,suest,brinkmann}@uni-mainz.de

Simon Pickartz, Stefan Lankes, and Antonello Monti
Institute for Automation of Complex Power Systems, E.ON Energy Research Center, RWTH Aachen, E-mail: {spickartz,slankes,amonti}@eonerc.rwth-aachen.de

Lars Nagel
Department of Computer Science, Loughborough University, E-mail: l.nagel@lboro.ac.uk

Keywords Virtualization · Checkpoint / Restart · Migration · HPC

1 Introduction

Supercomputers are moving towards exascale computing to fulfill the continually growing demands of HPC applications. Compared to today's systems, this performance gain will not only be achieved by an increase of the node count but also by a rising amount of cores per node. Exascale-ready applications are highly optimized to use all available cores of a single node, while many typical HPC applications running on similar architectures are able to scale to many nodes, but not to use hundreds of cores on a single node. They typically stress one specific resource on a compute node, such as the CPU, the memory, or the I/O subsystem. The sharing of nodes by multiple applications, i. e., the co-scheduling of applications with distinct resource demands, can overcome the resulting scaling limitations within a node. It has been shown that this approach can increase both the overall system utilization and the energy efficiency [6, 40]. However, as applications have varying resource demands over time, dynamic load balancing is required to avoid the congestions of single resources. According load balancers require migration support to move jobs or parts of them across the cluster.

Increasing the number of cores and nodes of the system introduces new issues. The increasing number of hardware components of the system decreases the mean time between failure (MTBF). While the MTBF was in the order of days (BlueGene/L, Nov 2005) [21], it will further decrease for exascale systems [3, 10, 16]. When errors can be detected in advance, then application migration can solve this problem and increase the system's resiliency. In the case of imminent failures, an evacuation of affected nodes can be performed by a migration of the respective processes [30, 44]. Application checkpointing is another way of solving this problem [22]. An application can save its status, i. e., checkpoint, at regular intervals to a reliable storage. The checkpoint can be triggered by the application or the system. In the case of a failure, the application can be restarted from its last checkpoint.

In previous studies we investigated different migration techniques and found full virtualization based on Kernel-based Virtual Machine (KVM) [24] to offer high flexibility while providing performance results comparable to a native execution [33] [5]. A drawback of this approach are high migration times caused by the transfer of partly unused and therefore unnecessary memory regions. This is due to lack of information on the system level performing the migration, i. e., only the application may distinguish between data necessary for its further execution and data that can be discarded prior to the migration. In the case of Virtual Machine (VM) migrations, this is aggravated by the additional level in the address translation that comes with full virtualization. The hypervisor is not capable of detecting memory that has been freed by applications running within the VM.

In this article we propose to accelerate application migration in HPC by a reduction of the transmitted data volume and for decreasing the storage size of HPC applications' checkpoints. We showcase our approach by taking the example of VM migration/checkpointing. The migration time of VMs is mainly determined by the network bandwidth [19] and the size of the virtual machine image comprising the guest operating system and the application's processes. For a reduction of the VM checkpoint image size and an acceleration of the migration, hypervisors apply compression [41] and zero-block detection [12]. We leverage these mechanisms to reduce the transmitted data to the minimum that is required to resume the VM on the target node and to further decrease the storage size of the VM checkpoint image.

When executed within a VM, the release of memory does not affect the amount of data that is transferred during a migration or saved in a checkpoint since these regions are only freed within the guest system but not returned to the host. The same holds for the runtime, i. e., the *glibc* preserves freed memory to serve further allocations during the course of the application's execution. Therefore, we overwrite these freed regions with zeros. This way, the zero-page detection and the compression algorithm are able to further reduce the amount of migrated/checkpointed data. In our approach, we substitute the memory operations *realloc* and *free* to place zeros in every freed memory region. We evaluate the approach by running a set of HPC applications from various domains within VMs based on KVM. Our results show, that the presented approach reduces the migration time by up to 10 % when it is applied alone and by up to 60 % when it is combined with compression. We show that the overhead of our approach is neglectable for most applications and that it reduces the checkpoint size of our tested applications by up to 9 % if compression is not enabled in the hypervisor and by up to 49 % if the hypervisor enables compression for our approach and for the baseline.

This article is an extended version of "Accelerating Application Migration in HPC" [15]. In comparison, it provides a much more detailed description of the methodology and evaluation. This includes a new in-depth analysis of the overhead induced as well as the benefits of compressing partial zero pages. In the new Section 5 we describe how the approach can avoid unnecessary zeroing operations and benefit migration / checkpointing best.

The remainder of this paper is structured as follows: After discussing related work in the following section, we detail our approach in Section 3. In Section 4 we present a comprehensive evaluation of our approach before generalizing our approach in Section 5 and concluding the paper in Section 6.

2 Related work

Application migration is used for fault tolerance and load balancing. Nagarajan et al. propose a fault tolerance scheme for MPI applications based on proactive migration [30]. They monitor the health of computing nodes for the detection of deteriorating behavior. This way they are able to anticipate node

failures. In such a case, the monitoring system triggers the migration of the node's processes to healthy nodes.

Application migration for load balancing is seldom exploited in HPC, but quite common in cloud computing. Load balancing strategies can include the current load distribution in the data center, historical data on the load and / or information about renewable energy [18] [29]. Randles et al. present a comparison of distributed load balancing strategies [36].

Migration techniques can be broadly divided into three categories: *process-level migration*, *virtual machine migration*, and *container-based migration* [33]. Process-level migration often leverages Checkpoint / Restart (C/R) mechanisms, which allow for capturing snapshots of the current application state comprising all necessary information for the later restart on the same or another node. The simplest approach is *system-level C/R* which performs a memory core dump. It can be implemented in kernel space (see BLCR [11]) or in user space (see DMTCP [1]). The advantage of system-level C/R is the transparency to the application. Furthermore, checkpoints can be taken at arbitrary points of the program's execution. However, these approaches result in relatively large checkpoints because they include data that is not required for restarting the computation.

Application-level C/R was introduced with the goal to reduce checkpoint sizes. However, this comes at the cost of an increased complexity and the involvement of the application programmer, who is in charge of collecting all data structures that are required for a restart of the application. This process can be simplified by special libraries and compilers. The Libckpt library, e. g., provides transparent C/R, but requires user directives that mark the checkpoints' locations and data [34]. Bronevetsky et al. provide a source-to-source compiler tool that automatically instruments the code to save and restore its own state. The tool coordinates C/R for parallel OpenMP [7] and MPI programs [38]. An approach to reduce the checkpoint size of application-level C/R, which is orthogonal to the approach presented in this paper, is to deduplicate different generations of checkpoints [23].

Compression can also be used to reduce the checkpoint size. Ibtesham et al. examine the feasibility of using compression to reduce checkpoint size in HPC environment [20]. Their study reveals that checkpoint compression is a potentially useful optimization for large-scale scientific applications. Islam et al. introduce data aware checkpoint compression to improve the compressibility of HPC applications' checkpoints and decrease the checkpointing overhead [21]. They compress multiple checkpoint files from different processes together. Data aware checkpoint compression extracts metadata semantic inside a process checkpoint file and then it uses this knowledge to merge the checkpoints parts from various processes intelligently. Often compression techniques have a finite window where they look for similarities. However, with the provided semantic data, similarities can be searched within all the checkpoint files.

The virtualization overhead is often seen as the main reason why virtual machine migration is rarely used in HPC. Youseff et al. and Birkenheuer et al. show that this overhead can be neglected and that the performance of virtual

machines is relatively close to native execution [45] [4]. Breitbart et al. demonstrate that the employment of virtual machines has only a minimal impact on the applications' performance in co-scheduling scenarios [5].

A lot of effort has been spent on the acceleration of VM migration, either focusing on increasing the bandwidth between source and destination nodes or on finding better algorithms for copying data between the nodes. None of the studies investigated what is contained in the virtual machine image and whether it is needed. Huang et al. propose a high performance virtual machine migration design that uses Remote Direct Memory Access (RDMA) over InfiniBand [19]. In this way, they are able to increase the available bandwidth for migration and reduce the migration overhead by 80 % compared to Gigabit Ethernet. Satyanarayanan et al. propose a Suspend / Resume (S/R) approach for virtual machines, in which a suspended virtual machine saves its volatile state to a file [26, 37]. This file is copied to a remote node where the virtual machine can be resumed.

Live migration is a technique for moving a VM between compute nodes with almost zero downtime. There are different techniques for live migration, for example, *precopy* and *postcopy*. Hirofuchi et al. propose live migration with postcopy in which the content of a virtual machine is copied after its process state has been sent to the target node [17]. Once the process state starts execution on the target node, virtual machine memory pages are fetched on demand from the source node. The precopy approach proposed by Clark et al. first copies the whole memory state of the virtual machine from the source to the destination node, while still running the virtual machine on the source node [8]. As already transferred memory might get updated after being copied from the source node, updated memory pages are iteratively copied to the destination node before finally the process state can be copied to the target node. Precopy works well for read-intensive workloads, while write-intensive applications accessing large amounts of memory can render this migration approach impossible [17]. Precopy is nevertheless more resistant to faults because the source node still holds an updated copy of the virtual machine and typically experiences shorter downtimes, as the migration data can be transferred in bigger chunks to the destination node.

3 Methodology

This section describes our approach to accelerate application migration. By taking the example of VM migration, we first provide an overview of the migration mechanism inside QEMU / KVM and then describe the preload library for zeroing freed memory which relies on the *GNU C Library* (or *glibc*) version 2.17. Finally, we briefly present the HPC application benchmarks that we use for the evaluation of our approach.

3.1 Virtual Machine Migration in QEMU / KVM

KVM is an open source Linux-based virtualization solution [24]. It provides full virtualization on x86 hardware utilizing the VT-x or AMD-V hardware extensions [42] [43] and is implemented as a loadable kernel module. Starting from Linux 2.6.20, the kernel components of KVM are part of the Linux main branch.

KVM only virtualizes the CPU and the memory subsystem in kernel space. Device emulation, e. g., access to virtual hard disks, and migration are performed by the user-space emulator QEMU [2]. QEMU supports *cold* and *live* migration. The former – often referred to as *stop-and-copy* migration – basically leverages checkpointing techniques. The guest VM has to be suspended to obtain a consistent state and afterwards this state is sent over the network to the target node. In contrast, live migration allows the guest to continue its execution during the migration process. A popular method is the *pre-copy* live migration which has been introduced by Clark et al. [8], which is split into two phases: (1) the push-phase in which the guest keeps running on the source host while its memory pages are already transferred to the target host. Since the guest may modify pages that are already transferred, theses have to be tracked and re-transmitted, i. e., the first phase is an iterative process that lasts until a certain termination criterion is met. (2) the migration finalizes with the stop-and-copy phase corresponding to the cold migration. Here, the guest is stopped on the source host and the remaining *dirty* pages are transferred to the target host. Since these are likely to be considerably less than in the case of a cold migration, the live migration is able to drastically reduce the guest's downtime at the expense of a higher network load.

The implementations of cold and live migration within QEMU are very similar and we therefore restrict the analysis to the former in the scope of this work. For an understanding of the underlying migration logic, a closer look into the implementation of VMs by QEMU / KVM is necessary. A VM is started as a normal process from the host's point of view. Therefore, QEMU allocates a region within the virtual address space representing the physical memory of the VM, i. e., the guest-physical memory. Just as with any other process, these memory pages are not backed by physical page frames before the process, i. e., in this case the guest system, modifies the according regions.

During the migration process, QEMU traverses this virtual memory region on the source node to determine which pages have to be transferred to the destination to successfully resume the VM. QEMU is purely implemented in user-space and therefore does not know the actual page mapping and whether a particular page has already been used by the guest or not. However, virtual pages that do not point to a physical page frame always point to the so-called `NULL` page – a page frame that solely contains zeros. This fact is leveraged by a zero-page detection algorithm within the migration logic. Each page is analysed during the memory traversal whether it only contains zeros. During the migration, QEMU omits these *zero-pages* as they are not required to successfully resume the VM on the target node. The zero-detection is imple-

mented by an unrolled loop that can easily benefit from vector operations. Furthermore, QEMU takes care of the proper handling of TCP / IP connections to / from the guest during migration, i. e., the migration is transparent from the application's point of view since the protocol handles packet losses and re-transmissions of missing packets.

Although the described zero-page detection already improves migration times, especially for guests with a low memory footprint, it only works for memory regions that have never been used by the guest. Memory regions that were used and then freed by guest processes cannot be detected without further effort. This is because a process running within the guest only returns the memory to the guest kernel, but it is not returned to the host. Hence, QEMU will still transfer these pages to the target node as they are likely to contain values different from zero. If the pages were returned to the host kernel, the according page mappings would again point to the `NULL` page. In that case, the zero-detection would be capable of detecting these regions and omit them during the migration process.

QEMU starting at version 2.4 also supports the migration of compressed VMs. If enabled, each RAM page of a VM is compressed prior to the migration and decompressed at the destination node. The performance of the compressed migration can be fine-tuned by modifying the parameters *compress-level* and *(de-)compress-threads*. Compression is of course only applied to non-zero pages.

3.2 Virtual Machine Memory Zeroing

Our *zeroing preload library* reduces the amount of data migrated by intercepting memory deallocation calls and placing zeros in the deallocated memory regions (cf. Fig. 1) before the memory is returned to the system. The new zero regions either result in zero pages which are left out during the migration or in partial zero pages which can be compressed more efficiently.

Allocated chunks Memory must only be overwritten when it is deallocated. Deallocations are performed by functions provided by the *glibc* library, which is dynamically linked to the application at runtime. Memory is deallocated by calling `free()` or `realloc()`, which changes the size of allocated memory by freeing an old memory section before allocating a new one. The implementation of our preload library intercepts all deallocation related calls and clears the corresponding memory.

Glibc generates for each memory allocation a so-called *allocated chunk*. Allocated chunks are cascaded after each other in memory. When an allocated chunk is freed, it is connected to other freed chunks using a double-linked list. An allocated chunk contains besides the requested memory also metadata that includes pointers to the next and the previous free chunk (only if this chunk is currently deallocated), the size of this chunk, the size of the previous chunk (if allocated) and three flags (cf. Fig. 2). The flags encode whether the current

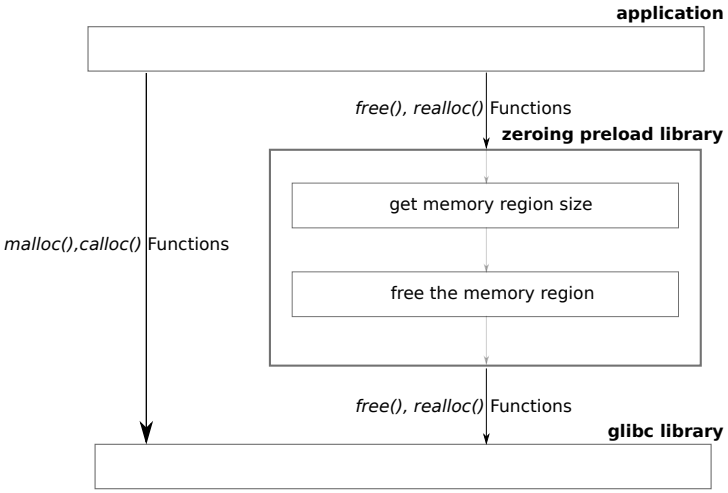


Fig. 1: The preload library intercepts deallocation operations issued by an application.

chunk is allocated via the `mmap()` system call, whether the previous chunk is in use and whether this chunk belongs to a thread arena (a separate heap memory which is maintained per thread).

The library only uses the application's pointer to the data and the metadata about the chunk size to zero the chunk's user data.

Zeroing Memory alignment and additional metadata are the reasons that the space occupied by an allocated chunk is larger than the memory requested by the user. It is important to only zero-out user data and not the corresponding metadata, as the metadata is still used by glibc, which moves the chunk to the double-linked list of free chunks once the memory is deallocated. We therefore carefully approximate the size of the user data by taking the chunk size minus 32 Bytes, which is an upper bound for the size of the chunk metadata [13].

It is necessary to intercept calls to the functions `free()` or `realloc()` before data can be zeroed. In order to do this, our library provides functions with the same signature (function name, number of arguments, and types of the arguments as well as of the return value) which are called instead of the original functions. The new functions call the old functions, but only after placing zeros in the user data region of the respective allocated chunk. The placement of the zeros is performed by calling `memset()`.

The proposed implementation does not require any extra metadata structures or the interception of allocation operations, so that its runtime overhead primarily depends on the number of deallocation operations, the size of deallocated memory and the durability of the operations. This overhead furthermore has to be small, so that the zeroing of regions, which get reallocated and written to again by the application, does not lead to increased runtimes. As shown

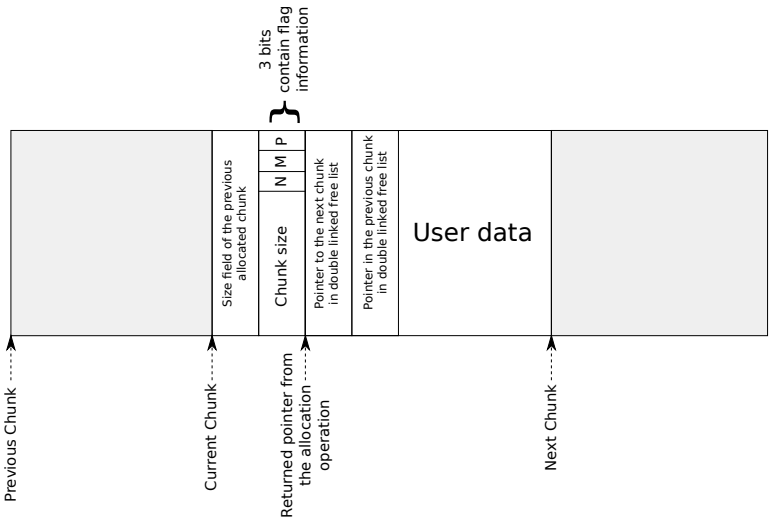


Fig. 2: The structure of an allocated chunk managed by glibc [13]. Only deallocated chunks are connected to a double linked free list.

in Section 4, the implementation of our library has a negligible overhead for most applications, while it can induce an overhead of 7% for selected applications like mpiBLAST.

3.3 HPC Application Benchmarks

We evaluated applications from different scientific areas like physics, chemistry, and biology. The following list briefly describes the five HPC applications, which have been used as benchmarks in our tests:

- NAMD is a parallel molecular dynamics simulator for large biomolecular systems. It can simulate more than a hundred million atoms utilizing up to 500.000 cores [31].
- mpiBLAST is the MPI-parallel version of the *Basic Local Alignment Search Tool* (BLAST). It is a sequence alignment tool that gets nucleotide / protein sequences as input, compares them to sequences in a database, and computes statistics about the matching results. mpiBLAST can scale up to hundreds of processors [9].
- GROMACS is a computational chemistry application that performs molecular dynamics simulations. It can solve Newton's equations of motion for millions of interacting particles [35].
- LAMMPS is a molecular dynamics simulator, which can also model atomic, polymeric, biological, metallic, granular, and coarse-grained systems using a variety of force fields and boundary conditions. It parallelizes the computation by spatially decomposing the simulation domain [14].

– PhyloBayes is a parallel implementation of the *Bayesian Markov Chain Monte Carlo* (MCMC) sampler for phylogenetic inference. The program uses nucleotide, protein, or codon sequence alignments to perform phylogenetic reconstruction [28].

4 Evaluation

In this section we evaluate our approach in terms of runtime overhead and checkpoint size. We show that the reduction of the checkpoint size results in a decrease of the migration time.

We used four NUMA systems for the evaluation. All systems have 32 virtual cores distributed over two sockets with 8 physical cores each. Two systems are equipped with Intel SandyBridge CPUs (E5-2650) and two with the newer generation Intel IvyBridge CPUs (E5-2650 v2). They are clocked at 2 GHz and 2.6 GHz respectively. The cluster is connected by a Gigabit Ethernet fabric and all nodes employ a software stack based on CentOS 7. The virtualization framework is based on KVM and QEMU version 2.5.1.

4.1 Zeroing Preload Library Overhead

The zeroing preload library introduces a runtime overhead which is mainly the time required to place zeros in the freed memory regions. The additional runtime therefore grows linear in the number of times memory is freed and the size of the affected memory regions, which are both highly application-dependent. To assess this overhead, we compared the execution time of each application with and without the preload library within a VM.

All benchmarks have been executed ten times and we took the median over these runs to obtain stable results. Figure 3 shows the execution times of our test applications which are normalized with respect to the median. The figure also includes the standard deviation. mpiBLAST exhibits a large preload library overhead of 7.1 %. After mpiBLAST comes PhyloBayes with overhead of 6.1 %. In contrast, GROMACS, NAMD, and LAMMPS, show a negligible overhead of less than 0.3 %. The negative value of −0.11 % for NAMD can only be explained by noise. Only PhyloBayes experiences a large standard deviation in comparison to the other applications. This can be explained by the randomness of the CAT model [39] used by PhyloBayes.

Finding: The runtime overhead of the preload library depends on the application and is at most 7.1 % for our sample applications.

For an assessment of the source of the overhead, we modified the zeroing preload library so that it generates a trace of the intercepted deallocation operations. The overhead is proportional to the amount of these operations and the size of the affected memory regions. The trace records the number of zero bytes placed in the deallocated memory regions and the number of

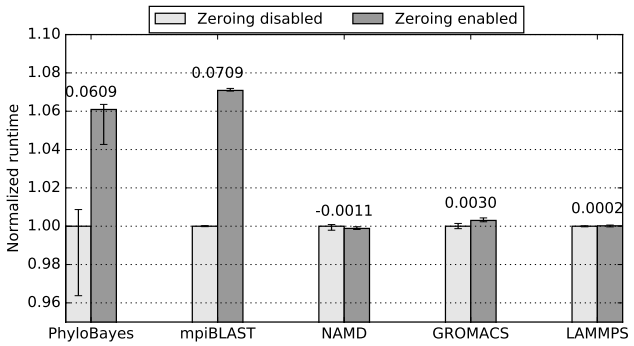


Fig. 3: Impact of the preload library on the runtimes of selected HPC applications. The bars represent the normalized execution time, the median, and the upper and lower quartiles. The numbers above the bars are the difference between the medians for enabled and disabled zeroing.

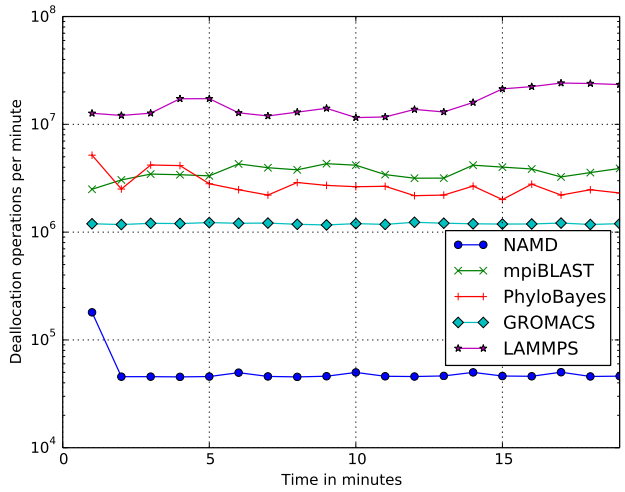
deallocation operations issued by the application at runtime. We ran each test application once inside of a VM with the modified preload library generating the trace. Figures 4a and 4b show the number of deallocation operations per minute and the amount of zeroed memory per minute. Every value in the graphs represents the accumulated amount of the previous minute.

mpiBLAST sets the biggest amount of memory to zero per minute (cf. Fig. 4b) and has the highest deallocation rate after LAMMPS (cf. Fig. 4a) which explains why mpiBLAST has the highest preload library overhead in Fig. 3. This is probably the result of a frequent usage of communication buffers within the MPI layer. After mpiBLAST comes PhyloBayes in the amount of memory set to zero per minute and the memory deallocation rate which explains why PhyloBayes has the second highest preload library overhead after mpiBLAST. LAMMPS, in contrast, reveals the lowest amount of memory set to zero per minute. For LAMMPS, some points are missing in Fig. 4b because the allocated chunk sizes were less than 32 Byte. Although the preload library intercepts every deallocation operation, it only places zeros in the deallocated memory if the size of the *allocated chunk* is greater than 32 Byte. As a result of this, no zeros were placed in these memory regions and hence the respective memory size is equal to zero, which cannot be plotted on the logarithmic scale.

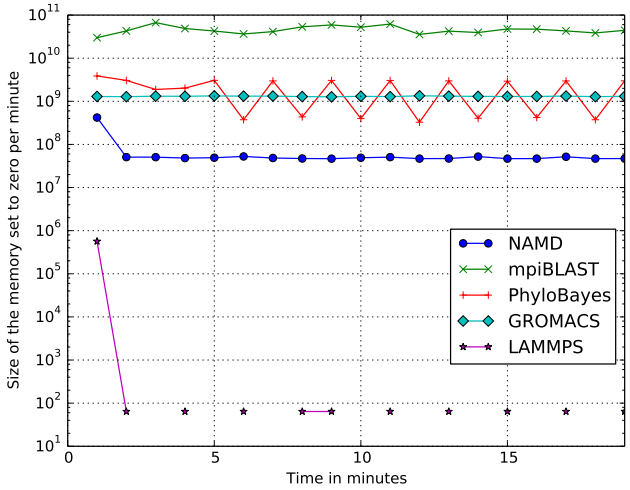
Finding: mpiBLAST has the highest overhead among the benchmark applications because it has the highest product of deallocation operations per minute and size of zeroed memory per minute.

4.2 VM Image Size

The migration time of an application running inside a VM is affected by the size of the VM's image which, in turn, mainly depends on the size of the ap-



(a)



(b)

Fig. 4: Evaluation of the preload library concerning (a) the number of deallocation operations intercepted by the library per minute and (b) the amount of data in bytes zeroed per minute.

plication's memory image. We studied the effect of our zeroing preload library on the size of the VM image and examined whether compression and the zero-page detection algorithm inside KVM benefit from it. Since KVM performs checkpointing as a memory core dump, this is a good measure for the VM's image size. Again, we ran each of our application benchmarks inside a VM with and without the zeroing preload library. We performed three checkpoints at intervals of 5 min and compressed the checkpoints logging the checkpoint size before and after compression. In the following, we always discuss the last of these checkpoints, while all other measurements have shown very similar results.

The zero-page detection algorithm is applied by the hypervisor to every checkpoint. It first discards complete zero pages and then applies compression to the remaining image (if compression is enabled). We extended this algorithms, so that is also logs the number of detected zero pages.

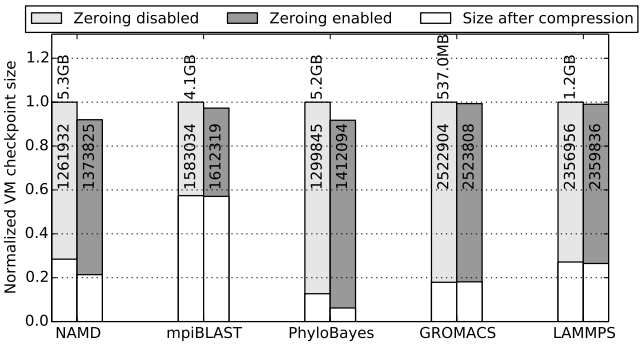


Fig. 5: Impact of the preload library on the checkpoint size. The values are normalized to the checkpoint size with disabled preload library and without compression. The numbers above the bars denote the respective absolute values. The numbers inside the bars refer to the amount of detected zero pages in the checkpoint image.

The summarized results comparing the combination of zero-page detection and compression with the standard approaches are shown in Fig. 5. The baseline is the image size without compression (zeroing disabled), where nevertheless already all pages are discarded, which only contain zeros. These results can be directly compared with the image sizes produced by our library (zeroing enabled), where deallocated memory blocks are set to zero. Observing these results, we see that, with disabled compression, all applications have a smaller checkpoint size when zeroing is enabled.

The bars in Fig. 5 for zeroing disabled and zeroing enabled also contain the image sizes after compression has been enabled. It can be seen that also in this case our strategy is able to decrease the image footprints for all applications

besides GROMACS. It is important to notice that the zeroing-library can have two positive effects on the resulting image size after compression has been enabled. First of all, all complete zero blocks are automatically removed by the hypervisor. Secondly, also pages, which only partially contain additional zeros can be more easily compressed. Especially the second effect is investigated in Section 4.4.

From the results we can derive that the zero-page detection algorithm is able to find additional zero blocks generated by the preload library. The reduction of the checkpoint size refers to the amount of the additionally detected zero pages and also to the amount of partially zeroed pages. For all applications except GROMACS, compression benefits from the additional partial zero pages generated by the preload library.

Finding: When compression is disabled, zeroing can decrease the checkpoint size by up to 9 %. Enabling zeroing and compression, the size can even drop by up to 94 % compared to the case of no zeroing and no compression. When compression is additionally enabled in the baseline approach, i. e., this corresponds to disabled zeroing but enabled compression, zeroing is able to decrease the checkpoint size by up to 49 %. The benefit of zeroing depends on the number of full or partial zero memory blocks detected at the time of the checkpoint.

4.3 VM Migration Time

One goal of our zeroing approach is to accelerate application migration. For this reason we compare the migration times of HPC application benchmarks with and without the preload library enabled.

We ran each application within a VM possessing 10 GiB of guest physical memory and migrated them back and forth between the two cluster nodes. The migration interval varied between 3 min to 5 min depending on the application runtime. The VM's virtual CPUs were mapped to the host's topology [5]. For doing so, we performed a one-to-one pinning of each virtual CPU to its counterpart on the host system, i. e., the CPU IDs seen by the guest match those on the host. Furthermore, the VM configuration comprises a virtual Non-Uniform Memory Access (NUMA) topology matching that of the host. Therefore, the virtual CPUs have to be grouped in so-called NUMA cells in accordance with the host's NUMA topology. As a result the guest system sees exactly the same hardware configuration as software running natively on the host. We used a Gigabit Ethernet link for data transfer and QEMU for the migration of the VMs with compression enabled or disabled and with default parameters, i. e., eight compression threads, two decompression threads, and a compression level of 1. To get stable results, we repeated this test 10 times and calculated the arithmetic mean. It is important to note that QEMU's zero-page detection algorithm discards full zero pages and that compression, if enabled, is only applied to the remaining pages.

As seen in Figs. 6a to 6j, the effect of the zeroing is highly application dependent. For NAMD, PhyloBayes, and mpiBLAST our approach accelerates the migration time regardless of whether compression was enabled or not. This is on par with the previous results estimating the impact of the preload library on the checkpoint size (cf. Fig. 5). Consequently, the zeroing does not have an impact on the migration performance of GROMACS and LAMMPS. When zeroing was applied alone without compression, NAMD benefited the most. It showed migration time savings of up to 10 % (cf. Fig. 6b).

The combination of zeroing enabled and compression enabled provided the least migration time for NAMD, PhyloBayes, and mpiBLAST. PhyloBayes benefited the most when zeroing and compression are enabled; its migration time was improved by up to 60 % compared to the case when migration is performed without zeroing and without compression (cf. Fig. 6d). Also, its migration time was improved by up to 19 % compared to the case when migration is performed without zeroing and with compression.

PhyloBayes and NAMD experienced a major improvement in the migration time saving when zeroing and compression are applied together comparing to the case when zeroing was used alone. On the other side, mpiBLAST only showed a minor improvement compared to PhyloBayes and NAMD. We relate this to the amount of partial zero pages injected by the zeroing library, which will be studied more closely in Section 4.4.

Although we only regarded the migration over Gigabit Ethernet, the presented approach might be interesting for other interconnects as well. In any case, the overhead generated by the preload library has to be compensated for by the savings that can be achieved with the given link speed.

Findings: The migration procedure of KVM benefits from the zeroing approach because it can leave out full zero pages and usually better compresses partial zero pages. The combination of zeroing and compression provides the best acceleration.

4.4 Partial Zero Pages and Compression

In this section, we investigate more closely when and how compression benefits from the zeroing approach. As mentioned before, full zero pages are discarded from the VM image before compression. Hence, compression is only applied to partial zero pages and full data pages. Of course, compression benefits from other data criteria, but they are not part of this study because we are only interested in zeros injected by our library.

In the previous test, in Section 4.3, mpiBLAST showed a minor improvement in the migration time saving when compression is added to zeroing compared to PhyloBayes and NAMD. mpiBLAST migration time saving was improved from around 6% to 29% (difference 23%) while PhyloBayes migration time saving was improved from around 4% to 60% (difference 56%) and NAMD migration time saving was improved from around 9% to 44% (difference 35%).

We relate this to the amount of partial zero pages in the VM image before compression.

To understand this phenomenon, we analyzed the effect of zeroing on the VM image. We re-ran the previous test from Section 4.3, but without compression, and instrumented QEMU to log the number of partial zero pages as well as the number and size of zero regions at every VM migration. The size of a continuous zero region in a page is always between 1 and 4095 bytes because the maximum page size is 4096 bytes and full zero pages are discarded. The resulting trace contains a histogram for each migration providing for each region size the number of occurrences of this region in the respective VM image.

To analyze the effect of the preload library on the zero distribution, we subtracted the histogram before the zeroing from the histogram after the zeroing. Then we computed the cumulative function of the result (cf. Fig. 7). The cumulative function accumulates the size of the differences in the partial zero pages. Figure 7 shows that the number of accumulated zeros increases for PhyloBayes and NAMD, but decreases for mpiBLAST.

In this figure, we are mostly interested in the final accumulated value at the largest region size, as it characterizes the total difference of zeros and therefore gives an indication about the compressibility of the image. PhyloBayes comes in first place, then NAMD, and finally mpiBLAST. For GROMACS and LAMMPS the final accumulated value of the difference in the partial zero pages is almost equal to zero. This result matches our expectations since the migration performance of both applications is not affected by the zeroing approach at all. Especially interesting are the results for mpiBLAST. Here, the accumulated value of zeros is even negative. This explains for the previous test, why mpiBLAST showed a minor improvement in the migration time saving when zeroing and compression are applied together compared to PhyloBayes and NAMD. This raises the question about why zeroing does not produce more zero ranges in the migrated image of mpiBLAST.

There are two possible reasons:

1. There are not many zeros placed, or their placement results in full zero pages, but not in partial zero pages. We observed this for mpiBLAST and our assumption is that deallocations of communication buffers are the main source for zero regions in the mpiBLAST image. As these buffers are often set to a multiple of the page size, the preload library only produces complete zero pages rather than additional partial zero pages.
2. Each application has different memory allocation and deallocation pattern. Zeros that our approach writes in the application memory might get reallocated and used by the application.

5 Generalization of the approach

In this section we discuss the current limitations of the memory management in the context of virtual machine migration / checkpointing. The main approach presented in this paper is to reduce the size of virtual machines by writing zeros

in freed memory regions which benefits zero-page detection as well as compression schemes. In this way, zero-page detection can discard complete zero pages for migration, and compression schemes achieve better results. However, freed memory regions that have been filled with zeros might be reallocated and reused by the application. This is, for example, common for communication buffers of MPI implementations. As these unnecessary zero writes only degrade the performance, our future goal is to completely eliminate them.

Figure 8 shows the memory allocation stack ranging from the application to the hardware level. The application allocates and deallocates memory using the functions *malloc()*, *free()*, *calloc()*, and *realloc()* of *glibc* which is a shared library in the application address space. It allocates and releases memory from the guest operating system using the system calls *brk()*, *sbrk()*, or *mmap()*. The guest operating system obtains memory from the host operating system through the virtualization layer.

Memory released by the application is returned to the *glibc* library. However, there is no guaranteed time period in which the *glibc* library returns the memory back to the guest operating system. In case of a VM, this is aggravated by the fact that the guest operating system never returns freed memory to the host operating system. One way to return this memory to the host operating system is to destroy the VM and to create a new one. Of course, this should only be done when the application running in the VM has terminated.

A better solution is to use our approach and enhance it by a small modification. The idea is to zero memory regions only when a migration or a checkpoint is imminent and thus to avoid unnecessary zero writes. The glibc library needs to be modified to react upon migration / checkpoint requests (similar to the approach for the migration of MPI applications in [32]). In doing so, it would place zeros into freed memory regions only directly before a migration starts. On the one hand, this would result in slightly higher migration / checkpoint latencies as the zeroing would add to the overall migration/checkpointing time. On the other hand, the runtime overhead would be reduced which is especially interesting for applications exhibiting a high memory allocation / deallocation frequency, e. g., as is the case for mpiBLAST.

In general, C libraries maintain the free memory regions by means of a buddy system [25] or similar data structures. Instead of zeroing the memory regions during migration / checkpointing, the C library is able to ignore free regions and may signal the hypervisor through a *hypercall* that the data should not be migrated. This approach would replace the zero page detection within QEMU and possibly yield an improved migration performance. However, this only affects free regions which occupy full memory pages. Partially freed pages benefit only if they are compressed and if zeros are written into them. Therefore, an ideal solution could combine both approaches: (1) freed memory regions smaller than the page size are filled with zeros while (2) full pages are completely omitted during the migration process.

Although we focused on VM migration in this work, the general concept may be beneficial to other means of process isolation, e. g., Operating Sys-

tem (OS) containers based on Docker[1], as well. The zeroing of freed memory regions affects two aspects of the migration procedure: (1) full zero-pages are omitted during the memory traversal; and (2) the compression benefits from partial zero-pages. The former aspect is specific to the migration of VMs since they reduce the amount of information available to the hypervisor, i. e., the mapping of the guest-physical memory to host-virtual memory is transparent. In contrast, the mapping of the virtual address space of OS containers is visible to the host OS, i. e., the concept of zero-pages becomes obsolete in this case. However, the second aspect likewise applies to container-based and even process-level migration. This is because these pages are part of the virtual address space of the isolation domain and therefore have to be transferred to the destination as well. In accordance with the observations made in Section 4.4, especially the zeroing of partial zero pages in conjunction with compression is able to improve the migration time significantly.

6 Conclusion & Future Work

In this paper, we have shown the limitations of the current memory management in the context of virtual machine migration / checkpointing and have presented a prototype to reduce the amount of data transmitted during virtual machine migrations and to decrease the storage size of virtual machine checkpoints. The approach places zeros in unused data regions such that zero-page detection and compression schemes work more efficiently. The evaluation reveals a positive effect on both, the migration time and the size of the application image. In particular, we demonstrated the merit of our method for HPC by choosing a set of test applications from this domain. Our approach reduced the migration time by up to 10 % when it is applied alone and by up to 60 % when it is combined with compression. We have shown that the overhead of our approach is negligible for most applications. We have also shown that our approach reduces the checkpoint size of our tested applications by up to 9 % without compression and up to 49 % with compression enabled in the hypervisor. This reduces the storage required for checkpoints.

In Section 5, we have explained how our prototype could be integrated in a general-purpose operating system with even smaller overhead in comparison to our current solution. In addition, the combination of specialized operating systems for virtual machines like Unikernels (e. g. HermitCore [27]) and our presented technique promises an additional reduction of the memory consumption and consequently shorter migration times and smaller checkpoint sizes.

In the future, we will explore the possibilities of virtual machine migration in different environments. Here we will especially focus on the challenges that appear with different networks and heterogeneous systems. Furthermore, we plan to integrate the developed migration techniques into the SLURM batch

[1] `https://docker.io`

system. Installed in a large cluster, we will analyze how the rescheduling of jobs can help to increase utilization and throughput. Scheduling virtual machines with respect to their long-time resource requirements raises new questions and research challenges.

We have made our implementation publicly available under `https://gitlab.rlp.net/brinkman/zeroingpreloadlibrary/`.

Acknowledgements This research and development was supported by the Federal Ministry of Education and Research (BMBF) under Grant 01IH13004 (Project FAST) and Grant 01IH16010B (Project Envelope).

References

1. Ansel, J., Arya, K., Cooperman, G.: DMTCP: Transparent checkpointing for cluster computations and the desktop. In: 23rd IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2009, Rome, Italy, May 23-29, 2009, pp. 1–12 (2009)
2. Bellard, F.: QEMU, a fast and portable dynamic translator. In: Proceedings of the FREENIX Track: 2005 USENIX Annual Technical Conference, April 10-15, 2005, Anaheim, CA, USA, pp. 41–46 (2005)
3. Bergman, K., Borkar, S., Campbell, D., et al.: ExaScale computing study: Technology challenges in achieving exascale systems peter kogge, editor & study lead (2008)
4. Birkenheuer, G., Brinkmann, A., Kaiser, J., Keller, A., Keller, M., Kleineweber, C., Konersmann, C., Niehörster, O., Schäfer, T., Simon, J., Wilhelm, M.: Virtualized HPC: *a contradiction in terms?* Software - Practice and Experience **42**(4), 485–500 (2012)
5. Breitbart, J., Pickartz, S., Weidendorfer, J., Monti, A.: Viability of virtual machines in HPC – A state of the art analysis. In: Euro-Par 2016: Parallel Processing Workshops - Euro-Par 2016 International Workshops, Grenoble, France, August 24-26, 2016, Revised Selected Papers, pp. 721–733 (2016)
6. Breitbart, J., Weidendorfer, J., Trinitis, C.: Case study on co-scheduling for HPC applications. In: 44th International Conference on Parallel Processing Workshops, ICPPW 2015, Beijing, China, September 1-4, 2015, pp. 277–285 (2015)
7. Bronevetsky, G., Marques, D., Pingali, K., Szwed, P.K., Schulz, M.: Application-level checkpointing for shared memory programs. In: Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2004, Boston, MA, USA, October 7-13, 2004, pp. 235–247 (2004)
8. Clark, C., Fraser, K., Hand, S., Hansen, J.G., Jul, E., Limpach, C., Pratt, I., Warfield, A.: Live migration of virtual machines. In: 2nd Symposium on Networked Systems Design and Implementation (NSDI 2005), May 2-4, 2005, Boston, Massachusetts, USA, Proceedings. (2005)
9. Darling, A., Carey, L., Feng, W.C.: The design, implementation, and evaluation of mpiBLAST. In: 4th International Conference on Linux Clusters: The HPC Revolution 2003 in conjunction with ClusterWorld Conference & Expo, pp. 13–15 (2003)
10. Dongarra, J., Beckman, P., Moore, T., et al.: The international exascale software project roadmap. Int. J. High Perform. Comput. Appl. **25**(1), 3–60 (2011). DOI 10.1177/1094342010391989
11. Duell, J.: The design and implementation of berkeley lab's linux checkpoint/restart. Tech. rep., Lawrence Berkeley National Laboratory (2003)
12. Dusser, J., Seznec, A.: Decoupled zero-compressed memory. In: High Performance Embedded Architectures and Compilers, 6th International Conference, HiPEAC 2011, Heraklion, Crete, Greece, January 24-26 2011. Proceedings, pp. 77–86 (2011)
13. Ferguson, J.N.: Understanding the heap by breaking it. Black Hat USA pp. 1–39 (2007)
14. FrantzDale, B., Plimpton, S.J., Shephard, M.S.: Software components for parallel multiscale simulation: an example with LAMMPS. Eng. Comput. (Lond.) **26**(2), 205–211 (2010)

15. Gad, R., Pickartz, S., Süß, T., Nagel, L., Lankes, S., Brinkmann, A.: Accelerating application migration in HPC. In: High Performance Computing - ISC High Performance 2016 International Workshops, Frankfurt, Germany, June 19-23, 2016, Revised Selected Papers, pp. 663–673 (2016)
16. Glosli, J.N., Richards, D.F., Caspersen, K.J., Rudd, R.E., Gunnels, J.A., Streitz, F.H.: Extending stability beyond cpu millennium: A micron-scale atomistic simulation of kelvin-helmholtz instability. In: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing, SC '07, pp. 58:1–58:11. ACM, New York, NY, USA (2007). DOI 10.1145/1362622.1362700
17. Hirofuchi, T., Nakada, H., Itoh, S., Sekiguchi, S.: Reactive consolidation of virtual machines enabled by postcopy live migration. In: Proceedings of the 5th International Workshop on Virtualization Technologies in Distributed Computing, VTDC@HPDC 2011, San Jose, CA, USA, June 8, 2011, pp. 11–18 (2011)
18. Hu, J., Gu, J., Sun, G., Zhao, T.: A scheduling strategy on load balancing of virtual machine resources in cloud computing environment. In: Third International Symposium on Parallel Architectures, Algorithms and Programming, PAAP 2010, Dalian, China, 18-20 December, 2010, pp. 89–96 (2010)
19. Huang, W., Gao, Q., Liu, J., Panda, D.K.: High performance virtual machine migration with RDMA over modern interconnects. In: Proceedings of the 2007 IEEE International Conference on Cluster Computing, 17-20 September 2007, Austin, Texas, USA, pp. 11–20 (2007)
20. Ibtesham, D., Arnold, D., Ferreira, K.B., Bridges, P.G.: On the viability of checkpoint compression for extreme scale fault tolerance. In: Proceedings of the 2011 International Conference on Parallel Processing - Volume 2, Euro-Par'11, pp. 302–311. Springer-Verlag, Berlin, Heidelberg (2012). DOI 10.1007/978-3-642-29740-3_34
21. Islam, T.Z., Mohror, K., Bagchi, S., Moody, A., de Supinski, B.R., Eigenmann, R.: McrEngine: A Scalable Checkpointing System Using Data-aware Aggregation and Compression. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC '12, pp. 17:1–17:11. IEEE (2012)
22. Jin, H., Ke, T., Chen, Y., Sun, X.H.: Checkpointing orchestration: Toward a scalable hpc fault-tolerant environment. In: Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid 2012), CCGRID '12, pp. 276–283. IEEE Computer Society, Washington, DC, USA (2012). DOI 10.1109/CCGrid.2012.61
23. Kaiser, J., Gad, R., Süß, T., Padua, F., Nagel, L., Brinkmann, A.: Deduplication potential of HPC applications' checkpoints. In: 2016 IEEE International Conference on Cluster Computing (CLUSTER), Taipei, Taiwan, September 12-16, 2016, pp. 413–422 (2016)
24. Kivity, A., Kamay, Y., Laor, D., Lublin, U., Liguori, A.: kvm: the Linux Virtual Machine Monitor. In: Proceedings of the Linux symposium, pp. 225–230 (2007)
25. Knowlton, K.C.: A fast storage allocator. Commun. ACM **8**(10), 623–624 (1965). DOI 10.1145/365628.365655
26. Kozuch, M., Satyanarayanan, M.: Internet suspend/resume. In: 4th IEEE Workshop on Mobile Computing Systems and Applications (WMCSA 2002), 20-21 June 2002, Callicoon, NY, USA, p. 40 (2002)
27. Lankes, S., Pickartz, S., Breitbart, J.: HermitCore – A Unikernel for Extreme Scale Computing. In: Proceedings of the International Workshop on Runtime and Operating Systems for Supercomputers (ROSS 2016), held in conjunction with 25th International ACM Symposium on High-Performance Parallel and Distributed Computing (HPDC 2016). Kyoto, Japan (2016)
28. Lartillot, N., Lepage, T., Blanquart, S.: Phylobayes 3: a bayesian software package for phylogenetic reconstruction and molecular dating. Bioinformatics **25**(17), 2286–2288 (2009)
29. Mäsker, M., Nagel, L., Brinkmann, A., Lotfifar, F., Johnson, M.: Smart grid-aware scheduling in data centres. Computer Communications **96**, 73–85 (2016)
30. Nagarajan, A.B., Mueller, F., Engelmann, C., Scott, S.L.: Proactive fault tolerance for HPC with xen virtualization. In: Proceedings of the 21th Annual International Conference on Supercomputing, ICS 2007, Seattle, Washington, USA, June 17-21, 2007, pp. 23–32 (2007)

31. Phillips, J.C., Braun, R., Wang, W., Gumbart, J.C., Tajkhorshid, E., Villa, E., Chipot, C., Skeel, R.D., Kalé, L.V., Schulten, K.: Scalable molecular dynamics with NAMD. Journal of Computational Chemistry **26**(16), 1781–1802 (2005)
32. Pickartz, S., Clauss, C., Breitbart, J., Lankes, S., Monti, A.: Prospects and challenges of virtual machine migration in HPC. Concurrency and Computation: Practice and Experience **30**(9) (2018). DOI 10.1002/cpe.4412
33. Pickartz, S., Gad, R., Lankes, S., Nagel, L., Süß, T., Brinkmann, A., Krempel, S.: Migration techniques in HPC environments. In: Euro-Par 2014: Parallel Processing Workshops - Euro-Par 2014 International Workshops, Porto, Portugal, August 25-26, 2014, Revised Selected Papers, Part II, pp. 486–497 (2014)
34. Plank, J.S., Beck, M., Kingsley, G., Li, K.: Libckpt: Transparent checkpointing under UNIX. In: USENIX 1995 Technical Conference on UNIX and Advanced Computing Systems, New Orleans, Louisiana, USA, January 16-20, 1995, Conference Proceedings, pp. 213–224 (1995)
35. Pronk, S., Páll, S., Schulz, R., Larsson, P., Bjelkmar, P., Apostolov, R., Shirts, M.R., Smith, J.C., Kasson, P.M., van der Spoel, D., Hess, B., Lindahl, E.: GROMACS 4.5: a high-throughput and highly parallel open source molecular simulation toolkit. Bioinformatics **29**(7), 845–854 (2013)
36. Randles, M., Lamb, D.J., Taleb-Bendiab, A.: A comparative study into distributed load balancing algorithms for cloud computing. In: 24th IEEE International Conference on Advanced Information Networking and Applications Workshops, WAINA 2010, Perth, Australia, 20-13, April 2010, pp. 551–556 (2010)
37. Satyanarayanan, M., Gilbert, B., Toups, M., Tolia, N., Surie, A., O'Hallaron, D.R., Wolbach, A., Harkes, J., Perrig, A., Farber, D.J., Kozuch, M., Helfrich, C., Nath, P., Lagar-Cavilla, H.A.: Pervasive personal computing in an internet suspend/resume system. IEEE Internet Computing **11**(2), 16–25 (2007)
38. Schulz, M., Bronevetsky, G., Fernandes, R., Marques, D., Pingali, K., Stodghill, P.: Implementation and evaluation of a scalable application-level checkpoint-recovery scheme for MPI programs. In: Proceedings of the ACM/IEEE SC2004 Conference on High Performance Networking and Computing, 6-12 November 2004, Pittsburgh, PA, USA, p. 38 (2004)
39. Si Quang, L., Gascuel, O., Lartillot, N.: Empirical profile mixture models for phylogenetic reconstruction. Bioinformatics **24**(20), 2317–2323 (2008)
40. Süß, T., Döring, N., Gad, R., Nagel, L., Brinkmann, A., Feld, D., Schricker, E., Soddemann, T.: Impact of the scheduling strategy in heterogeneous systems that provide co-scheduling. In: Proceedings of the 1st COSH Workshop on Co-Scheduling of HPC Applications, COSH@HiPEAC 2016, Prague, Czech Republic, January 19, 2016., pp. 37–42 (2016)
41. Svärd, P., Tordsson, J., Hudzia, B., Elmroth, E.: High performance live migration through dynamic page transfer reordering and compression. In: IEEE 3rd International Conference on Cloud Computing Technology and Science, CloudCom 2011, Athens, Greece, November 29 - December 1, 2011, pp. 542–548 (2011)
42. Uhlig, R., Neiger, G., Rodgers, D., Santoni, A.L., Martins, F.C.M., Anderson, A.V., Bennett, S.M., Kägi, A., Leung, F.H., Smith, L.: Intel virtualization technology. IEEE Computer **38**(5), 48–56 (2005)
43. Walters, J.P., Chaudhary, V., Cha, M., Jr., S.G., Gallo, S.M.: A comparison of virtualization technologies for HPC. In: 22nd International Conference on Advanced Information Networking and Applications, AINA 2008, GinoWan, Okinawa, Japan, March 25-28, 2008, pp. 861–868 (2008)
44. Wang, C., Mueller, F., Engelmann, C., Scott, S.L.: Proactive process-level live migration and back migration in HPC environments. Journal of Parallel and Distributed Computing **72**(2), 254–267 (2012)
45. Youseff, L., Wolski, R., Gorda, B.C., Krintz, C.: Evaluating the performance impact of xen on MPI and process execution for HPC systems. In: Proceedings of the First International Workshop on Virtualization Technology in Distributed Computing, VTDC@SC 2006, Tampa, Florida, USA, November 17, 2006, p. 1 (2006)

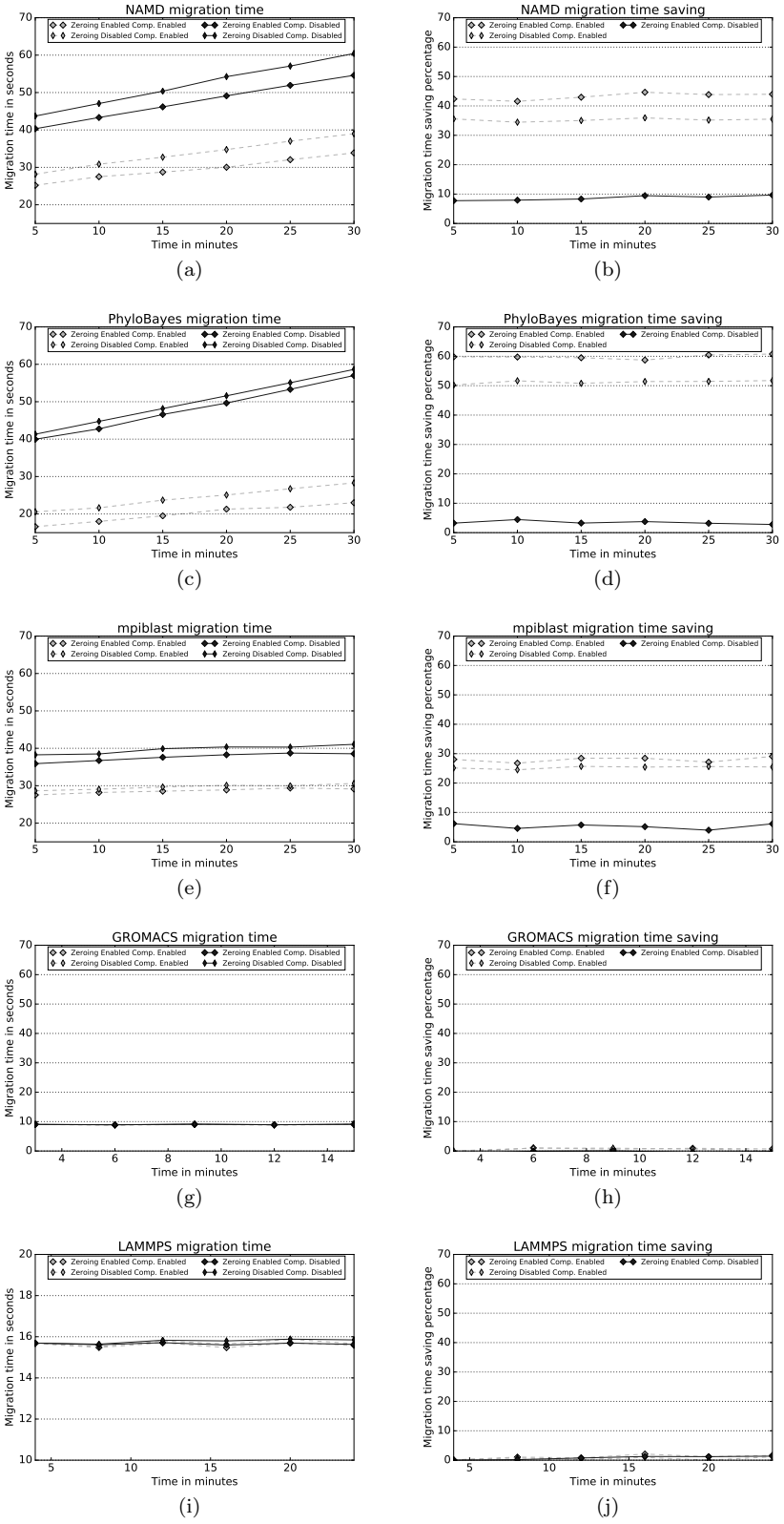


Fig. 6: Migration time and Migration time saving with zeroing enabled or disabled when compression is enabled (dashed curves) or disabled (solid curves): (a-b) NAMD, (c-d) PhyloBayes, (e-f) mpiBLAST, (g-h) GROMACS, and (i-j) LAMMPS. The saving is with respect to the case when zeroing and compression are disabled.

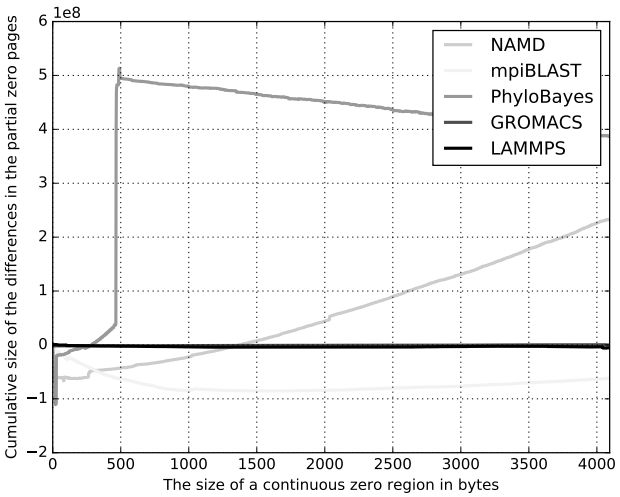


Fig. 7: Cumulative function of the volume of the difference in the partial zero pages in bytes. The difference is between the two cases when zeroing enabled and disabled. Only mpiBLAST shows a total negative accumulated value.

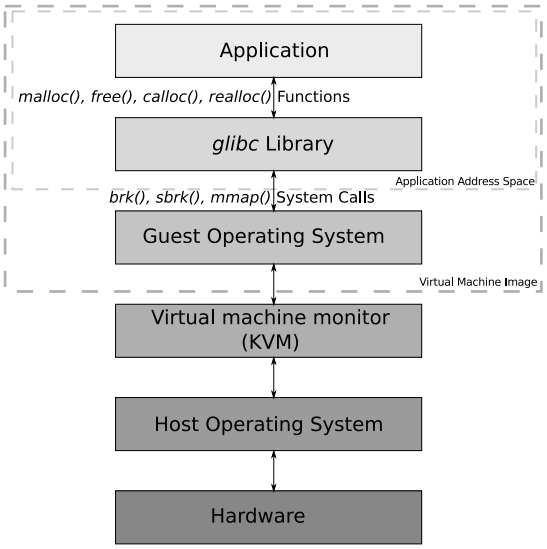


Fig. 8: Memory allocation stack. The application accesses memory using the *glibc* library which interacts with the guest operating system using system calls. This is a shared library in the application address space. The application and the guest operating system are included in the VM image.