Enhancing application checkpointing and migration in HPC

Dissertation submitted
for the award of the title
"Doctor
of Natural Sciences"
to the Faculty of Physics, Mathematics, and Computer Science
of Johannes Gutenberg University Mainz
in Mainz

Ramy Mohamed Aly Gad

Born in Alexandria, Egypt

Mainz, March 6, 2018

Abstract

It is predicted that the number of nodes and cores per node will rapidly increase with the upcoming era of exascale supercomputers. Such growing number of hardware components causes a decrease in the Mean Time Between Failures (MTBF). Furthermore, in order to efficiently exploit such exascale systems and fully utilize all available resources of a node, multiple applications could share execution on one node and compete for the resources available on this node, for instance, computing cores and accelerators. However, applications competing for the same resources, result in resource overload. Application checkpointing and migration are promising solutions to improve fault tolerance and to balance workloads between computing nodes, while avoiding resources overload.

In this thesis, we address part of the challenges related to performing application checkpointing and migration in HPC. We consider the problem of checkpointing for load balancing between different resources on a heterogeneous node. This problem is affiliated with context switching between the host and the accelerator memory spaces. We present a tool collection (*ConSerner*) (Context Serializer) that automatically identifies, gathers, and serializes the context of a kernel and migrates it to an accelerator's memory, where an accelerator kernel is executed with this data.

We consider the problem of reducing checkpoint size and migration time in a virtualized HPC environment. We notice that not all data objects within a virtual machine (VM) image are required for a successful checkpoint or a migration from a source to a destination node. Therefore, discarding these data objects that are not required from the virtual machine image before migration/checkpointing significantly decreases the migration time and the checkpoint storage size. In this thesis, we propose a novel approach for the acceleration of VM migration and the reduction of VM checkpoint storage size. We take advantage of the fact that freed memory regions within the guest system are not recognised by the hypervisor. Therefore, we fill them with zeros, so that zero-page detection and compression can work more efficiently. We demonstrate that our approach can boost the migration time by up to 10 %, when it is applied alone, and by up to 60 %, when it is combined with compression. We also show that our approach reduces the checkpoint size of our tested applications by up to 9 %, without compression, and by up to 94 % with compression.

Furthermore, for checkpointing, we consider the problem of scalability and the problem of checkpoint size reduction. A study on a wide range of HPC applications is performed. For each application, we show the deduplication potential of its checkpoints for different deduplication configurations. Since not all applications provide built-in checkpointing, we use DMTCP for system-level checkpointing. Using this type of checkpointing, we show that there is a high potential for saving data sent to disk and for increasing checkpointing performance and scalability.

Personal Publications

[GSB14] Ramy Gad, Tim Süß, and André Brinkmann. Compiler driven automatic kernel context migration for heterogeneous computing. In *Distributed Computing Systems (ICDCS), 2014 IEEE 34th International Conference on*, pages 389–398, June 2014.

[PGL+14] Simon Pickartz, Ramy Gad, Stefan Lankes, Lars Nagel, Tim Süß, André Brinkmann, and Stephan Krempel. *Migration Techniques in HPC Environments*, pages 486–497. Springer International Publishing, Cham, 2014.

[GPS+16] Ramy Gad, Simon Pickartz, Tim Süß, Lars Nagel, Stefan Lankes, and André Brinkmann. *Accelerating Application Migration in HPC*, pages 663–673. Springer International Publishing, Cham, 2016.

[KGS+16] Jürgen Kaiser, Ramy Gad, Tim Süß, Federico Padua, Lars Nagel, and André Brinkmann. Deduplication potential of hpc applications' checkpoints. In *2016 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 413–422, September 2016.

[SDG+16a] Tim Süß, Nils Döring, Ramy Gad, Lars Nagel, André Brinkmann, Dustin Feld, Eric Schröder, and Thomas Soddemann. Impact of the scheduling strategy in heterogeneous systems that provide co-scheduling. In *1st COSH Workshop on Co-Scheduling of HPC Applications, COSH@HiPEAC 2016*, pages 37–42, 2016.

[SDG+16b] Tim Süß, Nils Döring, Ramy Gad, Lars Nagel, André Brinkmann, Dustin Feld, Thomas Soddemann, and Stefan Lankes. Varysched: A framework for variable scheduling in heterogeneous environments. In *2016 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 489–492, September 2016.

[SDG+17] Tim Süß, Nils Döring, Ramy Gad, Lars Nagel, André Brinkmann, Dustin Feld, Eric Schricker, and Thomas Soddemann. *Impact of the scheduling strategy in heterogeneous systems that provide co-scheduling*, pages 142–162. IOS Press, 2017.

Contents

Chapter 1

Introduction

Computers are a remarkable invention of the twentieth century [Cer03]. They are becoming dominant in our life starting from embedded devices, to desktop computers, to large-scale supercomputers. The scientific and engineering applications of computers are numerous. They have allowed scientists to solve complex mathematical problems and perform fast data analysis which is impossible to be performed by human. Scientists can almost model everything with the help of computers from protein molecules to galaxies.

There is a need for faster computers. It is driven by the applications' demands for more computation power and the competition to solve more complex problems, to generate discoveries faster and more efficient with a reduced solution time from days or weeks to hours. A desktop computer cannot satisfy this need. This motivates the development of High Performance Computers (HPC). A High Performance Computer is a supercomputer that has a massive computation power (in term of Floating Operation Per Second (FLOPS)) and memory. Typically this supercomputer is composed of many computers called nodes. For space reason, sets of nodes are packed together to form a rack. Nodes are connected to each other using fast interconnect networks. There are software layers that allow communication between nodes, for example MPI [GL93]. The arrangement of these machines together is known as HPC cluster.

Bringing more computation power to satisfy the application needs is an interesting problem. From a hardware perspective, increasing the clock frequency is one way of solving this problem. However, increasing the clock frequency, increases the cooling cost and decreases the reliability. This is known as the power wall problem [VJO$^+$14, NWK05, Kur01].

Driven by Moore's law, which states the number of transistors will be doubled approximately every two years [Moo00], more and more gates are built on the same die. The transistor size is usually shrinking with every new technology. As more gates can be built per unit die area, more function units (for example, adders, subtractors, multipliers) can be added to the chip. By operating these function units in parallel, the throughput is increased. So parallelization is a one key to boost the computation power capability. Starting from a serial executing machine, instruction level parallelism is introduced by instruction pipelining [Pat11]. Techniques like branch prediction, speculative execution, out of order execution, are employed to improve the performance of the pipelining. Processors have vector instructions, such as, SIMD instructions, in which an operation is performed on multiple data at the same time [Pat11]. Multi-core processors are built on the same chip. Parallelization is also exploited by hyperthreading. It allows independent hardware threads executing on the same hardware core. These threads appear to the software as virtual computing cores. A computing node has multiple cores that operate in parallel and share the node memory.

From the software side, serial applications have to be transferred into parallel programming paradigms to use multiple computing resources in parallel. The parallelization can be performed on instruction level (SIMD instructions), thread level (OpenMP [DM98]), process level, (MPI [GL93]), or a mixture between all previous.

The continuous growing demand for more computation power has let to a competition between companies, countries, and institutes to build and own the fastest computer on Earth. The first petascale computer was "Roadrunner" by IBM, USA in 2008. Just three years later, Fujitsu built a Japanese ten petascale computer in 2011. At the time of writing this thesis, the fastest computer on earth is "Sunway TaihuLight" by Sunway, China in 2016. This machine offers a maximum performance of 93 PetaFLOP and 1.25 petabyes of memory.

Now supercomputers are on the edge of the exascale era [Top]. The upcoming exascale era raises new questions about the system architecture and new problems that might appear at that scale. The problems include efficient utilization of the computing nodes, workload balancing to avoid resource congestion, reliability and fault tolerance [BWT15, DBM+11].

Looking at today's systems, this computation power gain can be achieved not only by increasing the node count, but also by raising the number of cores per node. Efficient exploitation of such exascale systems requires full utilization of all available resources of a node, which includes computing cores and accelerators. Single applications typically stress one specific resource on a compute node, like the CPU, the memory or the I/O. The sharing of nodes by multiple applications can saturate all resources within a node. In previous work, it has been shown that co-scheduling multiple applications with different resource requirements on the same node can increase the overall system utilization and energy efficiency [SDG+16a, BWT15]. However, as applications have varying resource demands over time, they can compete for the same resource. This results in resource congestion. Application migration across computing nodes appears as a solution to this problem, where migration decisions are issued by a dynamic load balancer that monitors the computing nodes' performance.

For exascale, 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 Failures (MTBF). While the MTBF was in the order of days (BlueGene/L, Nov 2005) [IMB+12], it will further decrease for exascale systems [DBM+11, BBC+08, GCR+07]. When errors can be detected in advance, application migration is still proposed as a solution to 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 [WMES12, NMES07]. Application checkpointing is another way of solving this problem [JKCS12]. An application can save its status information, 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.

Application checkpointing and migration are therefore key solutions to solve problems in exascale environments. However, applying them at that large scale introduces new challenges. In this thesis, we address some of these challenges. For checkpointing, we consider the problem of scalability, the problem of checkpoint size reduction and the problem of checkpoint for load balancing on a heterogeneous node. Creating checkpoints for many processes puts high pressure on the storage backend and the network. For job migration, we consider the problem of flexibility and the problem of decreasing job migration time.

High throughput, energy-efficient accelerators like GPUs are promising to boost performance to reach exascale [SIL+15]. However, using them is still associated with some difficulties. In most cases, they require hardware specific code. They sometimes have separated memory space from their host systems. Switching between them requires time-consuming copy operations, reformatting/serializing the used data structures and complicated device-specific memory allocation [KDK+11]. Nowadays, a system scheduler does not assign a job to an accelerator unless the hardware specific code of this job is available and the data objects required for the execution of this job are predefined [SDG+17,SDG+16b]. These data objects should be copied to the accelerator memory before the job execution. These difficulties make accelerators hard to use from a programmer perspective, especially for legacy code, and inflexible to incorporate in the scheduling process. As a result of this, workload balancing between resources on a heterogeneous node is infeasible.

In this thesis, we address the problems associated with using accelerators, mainly the context switching problem between the host and the accelerator memory spaces, to perform load balancing on a heterogeneous node. This also includes determining and serializing the data objects that need to be copied to the accelerator memory before job execution. These data objects that are needed can be considered as a special type of checkpoint, where the checkpoint is generated for a part of the program, we call it a kernel. A kernel is a section of the program responsible for doing the computationally expensive part.

The approaches presented in this thesis are not limited to HPC. They can also be applied to other areas, such as, cloud computing.

Contribution In this thesis, we address challenges associated with application checkpointing and migration in HPC. Application checkpoint and migration are key solutions for the efficient utilization of the computing nodes, workload balancing to avoid resource congestion, reliability and fault tolerance.

One way to efficiently utilize all components within a computer node is to have several application share execution on the same node. In our contribution in Chapter 3, we show that even in the case of a heterogeneous single node, we are able to load balance between the different entities on that node [GSB14]. We address the context switch problem between the host and the accelerator memory spaces and the problems associated with using accelerators. They often have an isolated memory space from their host systems. Data objects required for an execution of a kernel need to be explicitly serialized and copied to the accelerator memory before the kernel execution. It is the task of a programmer to serialize and copy these data objects.

We present a tool collection (*ConSerner*) (Context Serializer) that automatically identifies, gathers, and serializes the context of a kernel and migrates it to an accelerator's memory, where an accelerator kernel is executed with this data. In this way, the accelerator can be used concurrently with the host system CPUs. This is done transparently to the programmer. Complex data structures (e.g., n-dimensional arrays, lists, trees, graphs) can be used without the need for manual modification of the program code by a programmer. Predefined data structures in external libraries (e.g., the STL's vector) can also be used as long as the source code of these libraries is available.

ConSerner saves the programmer the burden of writing error-prone serialization codes manually. In this way, it facilitates the use of accelerators. It is provided as runtime libraries and a compiler plugin that is partially embedded inside the LLVM compiler framework [LA04].

In Chapter 4, we propose an approach to accelerate virtual machine migration in the HPC context and to reduce the storage size of a virtual machine checkpoint [GPS^{+}16]. The idea behind our approach is that we have noticed that not all data objects within the virtual machine image are required for a successful checkpointing or a migration from a source node to a destination node. The migration time of a virtual machine is mainly determined by the network bandwidth [HGLP07] and the size of the virtual machine image comprising the guest operating system and the application's processes. Also, the storage size of a virtual machine checkpoint is determined by the size of the virtual machine image. Therefore, discarding data objects that are not required from the virtual machine image before migration/checkpointing significantly decreases the migration time and the checkpoint storage size.

For a reduction of the virtual machine image size and an acceleration of the migration, hypervisors apply compression [STHE11] and zero-block detection [DS11]. When an application is executed within a virtual machine, the deallocation 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 back to the host. They are included in the virtual machine image during migration/checkpointing. Therefore, we overwrite these freed regions with zeros. This way, the zero-page detection and the compression algorithms are able to further reduce the virtual machine image size.

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 virtual machines based on Kernel-based Virtual Machine (KVM) [KKL$^+$07]. We demonstrate that our 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 can be neglected for most applications. The overhead is due to intercepting deallocation operations and writing zeros in the virtual machine image. We also show that our approach reduces the checkpoint size of our tested applications by up to 9 % without compression and by up to 94 % with compression.

In Chapter 5, we propose to apply data deduplication as a solution to increase checkpoint scalability and decrease checkpoint size [KGS$^+$16]. In this chapter, we investigate the deduplication potential for a wide range of HPC applications checkpoints. For each application, we show the deduplication potential of its checkpoints for different deduplication configurations. The coupling of checkpointing and deduplication for applications, which have a high deduplication rate, can help to improve checkpointing scalability. Since not all applications provide built-in checkpointing, we use DMTCP [AAC09] for system-level checkpointing. Using this type of checkpointing, we show that there is a high potential for saving data sent to disk and for increasing checkpointing performance. Furthermore, we show where redundancies occur and how they can be exploited best.

Chapter 2

Application Checkpointing and Migration

Checkpointing is the process of saving the status of an application at a certain point in time, so that the application can be restarted from that point. It is extensively used to provide support for software fault-tolerance [YG07], playback debugging and process migration [WHV+95].

HPC applications have continuously increasing demands for much more performance. One way HPC systems address these demands is scaling up the number of nodes and cores per node to deliver the target performance. However, the increase in the performance of such systems does not come with an increase in the reliability. As the system scales up, the reliability goes down. While the Mean Time Between Failures (MTBF), the average time that a device functions before failure, for the current petascale systems is in the order of hours [SMM+12, SG07b], it is expected that it will further decrease for exascale systems [DBM+11, BBC+08, GCR+07].

This reliability issue makes checkpointing very crucial for such systems because it allows them to restore their work in case of failure. Though from another point of view, checkpointing itself is an overhead for an application. The application execution has to be interrupted for saving its status. The checkpointing operations consume CPU cycles, memory, disk bandwidth and parallel file system (PFS) bandwidth, even the saved checkpoints might not be used for restoring status.

A carefully designed checkpointing scheme tries to make the checkpointing overhead as low as possible by decreasing the checkpoint size and time. Many factors affect the design choice. For example; whether the system or the application programmer performs the checkpoint, and how often checkpointing is done? When checkpointing is done too often, it introduces much overhead. It becomes a waste of resources that leads to poor performance. Checkpointing less frequently wastes application computation in a case of a failure which also leads to poor performance. In the best case, the checkpointing frequency should be tuned to the expected failure rate for a certain failure coverage [MMBdS14].

The checkpoint design scheme has to consider the additional challenge required for checkpointing multi-process applications. These applications usually have many processes which communicate with each other using message passing like MPI [GL93]. Checkpointing these processes is more complicated than just checkpointing every participating process alone. If every participating process takes its checkpoint independently, there is no guarantee that some in-flight messages might not get lost while checkpointing [FWL+14, GC11, EAWJ02].

Migration is the process of moving an application's execution from one computing resource to another [PGL+14]. It is used to support load balancing [SDG+16b, GSB14] and fault tolerance [WMES12, NMES07]. The migration can be performed locally between resources within the same node, for instance, migrating computation between a host and an accelerator in a heterogeneous node [GSB14]. Migration can also be used to transfer computation to an external node, for instance, virtual machine migration [BBK+12, YWGK06]. A carefully designed migration scheme must provide enough resources on the destination node for a successful migration. This is known as the residual dependency problem [MDP+00].

This chapter provides background to application checkpointing and migration, it is organized as follows. In Section 2.1, checkpointing is classified into two different flavors according to how they are performed; system-level and application-level checkpointing. Then, in Section 2.2, we discuss techniques which are used to enhance the checkpointing performance. These techniques either to decrease the checkpoint size, to increase the available checkpoint write bandwidth, or to avoid blocking the application while checkpointing. Checkpointing challenges of parallel multi-process applications are discussed in Sections 2.3. Applications of checkpointing which include fault-tolerance, playback debugging and migration are discussed in Section 2.4.

Finally, in Section 2.5, we end this chapter with a brief discussion about application migration. We discuss four different groups: process-level migration, task migration, virtual machine migration, and container-based migration.

2.1 Checkpointing

Checkpointing can be classified into two main types: system-level and application-level checkpointing.

2.1.1 System-level

The simplest checkpoint is a memory core dump of the computation status of an application to a file; it is performed at the system-level. The computation status of an application includes the content of the processor registers, the heap, the stack and the code region. The advantages of this approach are: the checkpoint is completely transparent, i. e. the application is unaware that it is being checkpointed. Checkpointing can be done at any arbitrary location. A checkpointing decision can be triggered from inside or outside the program. The disadvantages of this approach are: it produces a relatively large checkpoint in comparison to application-level checkpointing. As the system-level checkpointing is a memory core dump, everything is included in the checkpoint even if it is not required for the restarting process. However, system-level checkpointing is still attractive for legacy code and for the cases where application-level checkpointing is not supported by the application.

System-level checkpointing is divided into kernel-level and user-level checkpointing.

Kernel-level (OS provided checkpointing)

Kernel-level checkpointing is implemented inside the kernel space. The advantage of kernel-level checkpointing is that it provides better transparency. There is no need to replicate kernel data structures in user-space. Kernel-level checkpointing is able to restore the process ID and section ID of a process. This feature is desirable for applications that may rely on their parent's and/or children's PID to remain constant. For example, consider a parent process waits for one of its child processes to exit. The parent process keeps a track of its child process using the child's PID. On checkpoint restart, if the child's PID is different from the original (i. e. the PID before restart), the parent process will wait forever(see Figure 2.1).

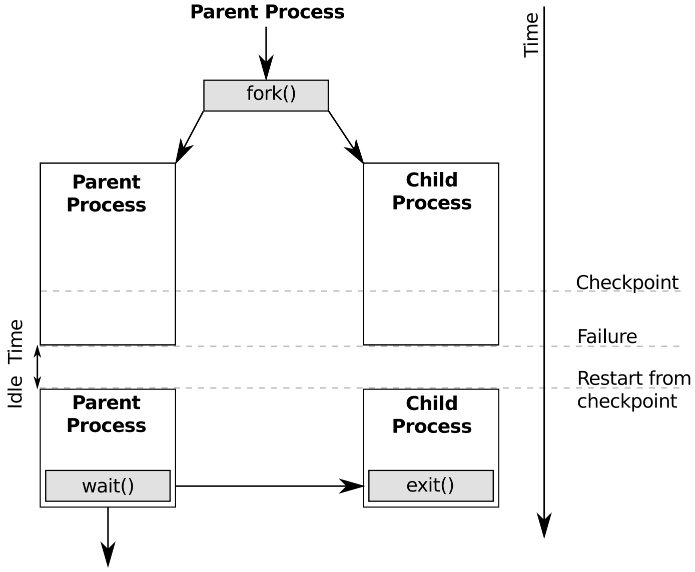


Figure 2.1: A block diagram of a parent process waiting for its child process to exit. The parent will wait forever if the child's PID changes after a restart.

The disadvantage of kernel-level checkpointing is that it requires kernel modification. This modification is usually shipped as a kernel module. The kernel module has to match the kernel version [Kim05]. Newer versions of Linux kernels require new kernel modules to be developed which makes kernel-level checkpointing tools hard to maintain for future use. Kernel-level checkpointing is less portable because it requires administrative rights to be installed which is not usually possible for normal users.

Berkley Lab Checkpoint / Restart for LINUX (BLCR) [Due03] is a kernel-level checkpointing tool. Initially, it was only providing support for single process applications. However, later some Message Passing Interface (MPI) libraries like OpenMPI uses BLCR as a plugin to support distributed checkpointing. BLCR supports up to Linux kernel version 3.7.x; it does not support newer kernel versions.

User-level

User-level checkpointing is implemented in user space. It is not dependent on a certain kernel version, which makes it much more portable. Therefore, no administrative rights are required to install or run the user-level checkpointing tool.

Libckpt [PBKL95] is a transparent system-level checkpointing tool that operates in user space. It implements an incremental checkpointing technique, in which only differences between successive checkpoints are saved instead of saving the entire checkpoints. Libckpt does not suspend an application execution while checkpointing as it performs a forked checkpoint technique. This technique clones the application status then performs checkpointing on this cloned status without interrupting the application execution. The advantage of forked checkpointing is that it minimizes the checkpointing time overhead. However, it requires the availability of enough memory on the compute node in order to duplicate the application status. Libckpt also proposes a user-directed checkpointing technique; this assumes that a little information from a programmer can yield large improvements in the checkpoint performance. The programmer provides some hints: to decide data objects that are excluded or included in the checkpoint and to choose the checkpoints locations.

Distributed MultiThreaded Checkpointing (DMTCP) [AAC09] is a transparent user-level checkpointing tool for distributed applications. Its novelty rests on its particular combination of features; It supports multi-threaded and parallel MPI applications. It operates in the user space and provides fast checkpoint times with negligible runtime overhead while not checkpointing. Ansel et al. show that, on 128 distributed core (32 nodes), a typical checkpoint time is 2 seconds or 0.2 seconds when forked checkpointing is enabled [AAC09]. We rely on this tool in Chapter 5.

2.1.2 Application-level

In application-level checkpointing, checkpointing is hard coded inside the application as it is implemented by the application programmer. The advantage of this approach is that it usually provides smaller checkpoint sizes than system-level checkpointing. The programmer uses his application knowledge to determine the minimum set of objects that should be included in the checkpoint and to decide the optimal checkpoint locations. The disadvantage of this approach is that the burden of the checkpointing becomes the application programmer's responsibility.

Some checkpoint classifications sort application-level checkpointing as user-level checkpointing because application-level checkpointing operates in the user space [GSjP05].

2.2 Checkpointing Enhancements Techniques

In this section, we give an overview of techniques that improve checkpointing performance. All the techniques aim to minimize the checkpointing time overhead and storage size.

2.2.1 Incremental Checkpointing

Instead of saving the entire application status, incremental checkpointing saves only the part of the application status that has been changed since the last checkpoint [GSjP05]. This requires a mechanism for determining what has been changed.

For full state checkpointing, a single checkpoint is sufficient to restore the application status. However, with incremental checkpointing, care must be taken with old checkpoint files. Restoring from a checkpoint requires keeping the old checkpoint files, which would increase the total storage requirements per checkpoint in comparison to full state checkpointing [Fro13]. So, occasionally a full status checkpoint is taken with incremental checkpointing.

The data block granularity on which the incremental checkpoint is performed affects the checkpoint efficiency. A large granularity makes incremental checkpointing useless because its size becomes almost equal to full checkpointing. A small granularity produces a small checkpoint, but it increases the overhead. So a compromise between both extremes has to be chosen. When incremental checkpointing is performed at the system level, the granularity is usually chosen equal to the page size [VMHR11].

Figure 2.2 shows an example of incremental checkpointing carried out on the system level. Only written pages and newly added pages are included in the incremental checkpoint. A single bit change in a memory page, forces the complete page to be included in the checkpoint. The old version of already written pages and deleted pages are marked to be deleted.

Mehnert-Spahn et al. propose a system-level incremental checkpointing based on write bit (WB) page protection tracking [MSFS09]. In their approach, the WB of every memory page is set to zero at every checkpoint. At the next checkpoint, the WB is checked whether it is set to one or not. If the application writes to a memory page whose WB is cleared, the kernel generates a page fault and the WB of this page will be set to one. The WB acts as an indicator whether a memory page has been written between the checkpoints or not. The advantage of this approach is that WB modification does not interfere with Linux page management functions like, page caching. The disadvantages of this approach are that it requires tracking the virtual memory area because pages might get mapped or unmapped between checkpoints.

Stertz et al. propose another approach for system level incremental checkpointing [Ste03]. This approach also detects writing on memory pages but by using dirty bit (DB) page tracking. It uses a kernel patch to shadow the DB information from the kernel within the user level and captures the modification status of the pages.

Vasavada et al. compare between write bit (WB) page protection tracking and dirty bit (DB) page tracking approaches for incremental system level checkpointing [VMHR11]. They have provided an optimized implementation of both approaches in BLCR [Due03] (a kernel-level checkpointing tool). In their evaluation, they favored the dirty bit page tracking approach as it has the potential to reduce the kernel activity significantly.

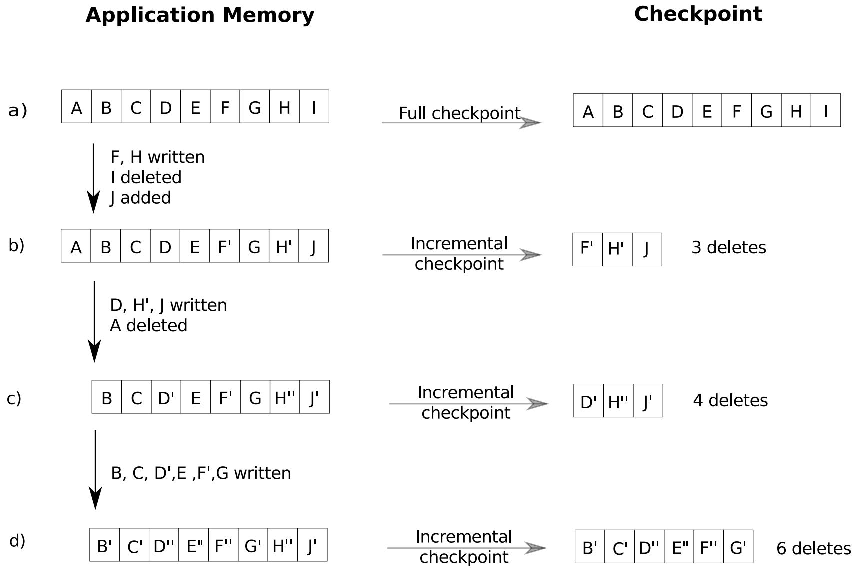


Figure 2.2: An example of incremental checkpointing carried out on the system level. Only written pages and newly added pages are included in the checkpoint.

2.2.2 Compression

Checkpoint compression is a practical solution to improve checkpoint performance. However, it is only useful when its benefit exceeds its cost. The cost includes the compression time and resources, for instance, CPU usage and memory. The compression time can be neglected when there is a slow disk and/or network bandwidth. Ibtesham et al. express this in Equation 2.1 [IAB+12]. Compression is only useful when:

$$\textit{Compression time} \quad + \quad \begin{array}{c}\textit{Time to save the}\\ \textit{compressed checkpoint}\end{array} < \begin{array}{c}\textit{Time to save the}\\ \textit{uncompressed}\\ \textit{checkpoint}\end{array} \tag{2.1}$$

which is equivalent to:

$$\frac{|checkpoint|}{Compression\ speed} + \frac{(1 - Compression\ factor) \times |checkpoint|}{Saving\ speed} < \frac{|checkpoint|}{Saving\ speed} \tag{2.2}$$

where $|checkpoint|$ is the checkpoint size. *Compression speed* is how much data can be compressed per unit time. *Saving speed* is the rate of checkpoint writes, i. e. how much checkpoint data can be saved per unit time. $Compression\ factor$ is the percentage of size reduction due to compression. Equation 2.2 can be reduced to Equation 2.3.

$$\frac{Saving\ speed}{Compression\ speed} < Compression\ factor \tag{2.3}$$

This means that, if the ratio of the checkpoints saving speed to the compression speed is less than the compression factor, checkpoint compression is beneficial. It then saves time and storage space.

Ibtesham et al. demonstrate that since the current computational power has increased at a faster rate than disk and network bandwidth, checkpoint compression can allow trading fast CPU workloads for slow disk and network bandwidth [IAB$^+$12]. More computation power means increasing the compression speed which decreases the size of the left-hand side in Equation 2.3. Plank et al. use compression with asynchronous checkpointing to lower the Parallel File System (PFS) checkpointing overhead [PL94]. Ibtesham et al. examine the feasibility of using compression to reduce checkpoint size in HPC environment [IAFB12]. 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 [IMB$^+$12]. 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. This approach is similar to our contribution in Chapter 5. However, in our case, we find similarities between processes' checkpoints using data deduplication which is considered as a special type of compression. In our case, similarities between processes' checkpoints are discarded by comparing the fingerprints of the chunks instead of relying on metadata semantic in the checkpoint files.

2.2.3 Zero block detection

As system level checkpoint takes a snapshot of an application's memory, zero blocks in the application's memory appear in the checkpoint image. Applications have a significantly large number of zero blocks resident in their memory. Ekman et al. show that the SPEC CPU 2000 benchmarks, on the average, the zero blocks represent 30% of the memory sizes [ES05]. They used a block size of 64 bytes. Dusser et al. confirm these results by performing the same experiment on the SPEC CPU 2006 benchmarks. They notice that the number of zero blocks is quite high exceeding 80% in several cases [DS11].

Our results in Chapter 4 confirm the finding of Ekman et al. and Dusser et al. We show that the zero blocks represent 10% to 92% of the applications' checkpoints. In our case, the block size is 4 KB, and we use a different set of parallel HPC applications.

Zero block detection is considered as a compression technique, in which the checkpoint image is divided into equal blocks, then these equal blocks are searched for complete zero blocks [DS11, MB09, ES05]. The found zero blocks are discarded from the checkpoint image, which in turn decrease the checkpoint storage size. Some metadata is needed to keep references to the discarded zero blocks. On checkpoint restart, the metadata is used to reconstruct the checkpoint image and to restore the discarded zero blocks. The size granularity of the block is tied to a particular implementation as it also affects the storage size of the metadata.

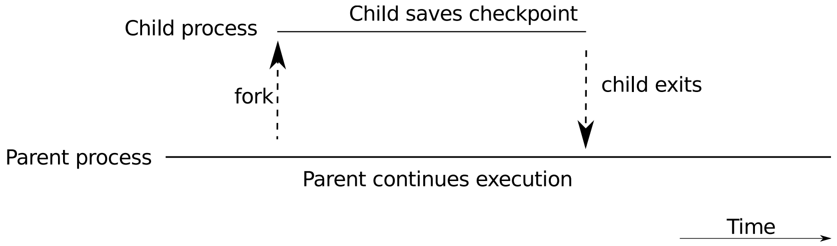


Figure 2.3: A block diagram of forked checkpointing.

2.2.4 Forked Checkpointing

With traditional checkpoint schemes, the application execution is interrupted while checkpointing is in progress. Once checkpointing is done, the application continues execution. However, with forked checkpointing, there is no need to interrupt the application execution [PBKL95,Vai95b]. A copy of the application state is made using the Unix *fork()* operation; then another asynchronously executing process is responsible for copying this application state copy to a file. In this way, the application execution and checkpointing can overlap. Figure 2.3 shows a block diagram of forked checkpointing. The parent process is not blocked by the checkpointing operation.

The advantage of the forked checkpointing approach is that it decreases the time overhead of the checkpointing process (reference to the wall clock time). However, it requires the availability of enough memory on the computing node to duplicate the application status.

Most operating system implement process cloning with copy-on-write [LNP94], in which a process and its clone share a memory page until one of them change this page. This memory page sharing decreases the memory required to perform the clone operation. The required memory is less than twice the memory allocated by the original process.

2.2.5 User-directed Checkpointing

User-directed checkpointing assumes that a little information from a programmer can yield large improvements in the checkpoint performance. This technique is used in system-level checkpointing tools so that they can get some of the information already available for application-level checkpointing. Both user-directed and application level checkpointing rely on the programmer information to improve the checkpoint performance. User-directed checkpointing is integrated into the system-level checkpointing tool Libckpt [PBKL95]. There are two ways in which the user information can boost performance, namely memory exclusion and synchronous checkpointing.

Memory exclusion

The user provides directives for excluding some memory locations from the checkpoint [PCL+99]. There are two cases where memory locations can be excluded. The first case is when the memory locations are dead, i. e. they are not going to be read or written in the future. The second case is when

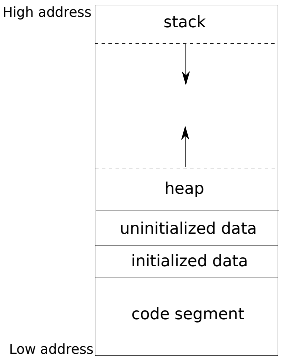


Figure 2.4: Memory organization of a typical program [Fer10].

the memory locations are clean, i. e. they are not being written since the last checkpoint. This case is similar to incremental checkpointing. However, in incremental checkpointing, the memory exclusion is done automatically without user direction.

A running program has four types of memory: code segment, data segment, heap, and stack [Fer10]. Figure 2.4 shows the memory organization of a typical program. The code segment contains the executable code. The data segment has two parts: The first is the initialized part where all the global, static and constant data are stored. The second is the uninitialized part which is set to zero when the program is first loaded.

A program allocates memory dynamically at runtime from the heap. For each new allocation, the heap grows in the direction shown in Figure 2.4. The lifetime of objects in the heap is either managed by the program with explicit allocation and deallocation functions or by a garbage collector [JHM16]. The stack stores a function local variables, arguments passed to that function and the return address of the instruction which will be executed after that function finishes execution. The stack grows in a last-in-first-out manner in the direction shown in Figure 2.4, when a function starts execution it adds its data on the top of the stack. Then when it finishes execution, it is responsible for removing that data from the stack. This mechanism helps the checkpointing tool in determining the lifetime of local variables. However, this is not the case for the program heap variables and for variables statically allocated in the data segment, i. e. global variables. The user-directed memory exclusion is quite useful in the case of the heap and global variables.

User-directed memory exclusion can reduce the checkpoint size of system-level checkpointing tools, but it needs to be used carefully. If a live memory region is mistakenly excluded from a checkpoint, this can cause the application to fail on recovery or produce incorrect results.

Synchronous Checkpointing

Synchronous checkpointing is a user directive that allows the programmer to choose the locations where the program should be checkpointed. The programmer should use it wisely with the memory exclusion directive. He should choose checkpoint locations where memory exclusion has the greatest effect. It is called

”synchronous checkpointing” because it is not initiated by timer interrupts.

Synchronous checkpoints may be placed in program locations that are reached more often. Checkpointing more often can lead to poor performance. To avoid this, some libraries, for example, Libckpt, require a minimum time interval between the checkpoints to be specified.

Listings 2.1 and 2.2 show a sample scientific program before and after adding memory exclusion and synchronous checkpointing directives [PBKL95]. The program reads some data *(D)* from an input file, processes the data, then writes the processed data to an output file. It performs this operation in a loop. In Listing 2.2, by choosing line 13 to carry out the checkpoint, *(D)* is discarded from the checkpoint. If *(D)* is large enough, user-directed checkpointing will be responsible for a significant saving in the checkpointing overhead. In line 14, *(D)* is included again because it is going to be used again starting from line 8.

Listing 2.1: Typical scientific program [PBKL95] that reads some data from an input file, processes the data, then writes the processed data to an output file. It performs this operation in a loop.

```
main()
{
 struct data *D;
 FILE *fi, *fo;
 D = allocate_data_set();
  fi = fopen(”input”, ”r”);
  fo = fopen(”output”, ”w”);
 while(read_data(fi, D) != -1)
 {
   perform_calculation(D);
   write_results(fo, D);
 }
}
```

Listing 2.2: The sample program in Listing 2.1 after adding memory exclusion and synchronous checkpointing directives [PBKL95].

```
   ckpt_target()
 2 {
    struct data *D;
 4  FILE *fi, *fo;
    D = allocate_data_set();
 6   fi = fopen(”input”, ”r”);
     fo = fopen(”output”, ”w”);
 8  while(read_data(fi, D) != -1)
    {
10    perform_calculation(D);
      write_results(fo, D);
12    exclude_bytes(D, sizeof(struct data), CKPT_DEAD);
      checkpoint_here();
14    include_bytes(D, sizeof(struct data));
    }
16 }
```

2.2.6 Diskless Checkpointing

An application checkpoint can be stored to a stable storage, i. e., disk. The stable storage offers a high level of resiliency; the application can be restored in case of complete node power failure. Since application checkpoints can be large (up to hundreds of megabytes per process) [KGS+16], storing checkpoints to a stable storage becomes a major contributor to the checkpoint overhead, i. e., performance degradation due to checkpointing. This performance degradation is noticeable in parallel and distributed systems where the number of running processes outnumbers the number of disks.

Diskless checkpointing reduces overhead by avoiding storing checkpoints to stable storage [PLP98]. Instead, it caches checkpoints in local node memory or other neighbor node storage. Diskless checkpointing enhances checkpointing by removing the disk. However, this does not come for free. Failure coverage of diskless checkpointing is less than stable storage checkpointing because components of diskless checkpointing might not survive a complete cluster power outage. Diskless checkpointing can recover from a single process failure and in some cases from multiple processes failure.

Vaidya et al. combine diskless checkpointing with disk-based checkpointing to build a two-level checkpointing scheme [Vai95a]. Diskless checkpoints are frequently taken to handle the most common one or more processes failure. Disk-based checkpoints are taken less often to survive the rare power outage failures.

Diskless checkpointing can be combined with incremental checkpointing and forked checkpointing to decrease the memory usage. With incremental checkpointing, only the difference between the checkpoints is saved. The difference includes the updated read/write pages. Read only pages are saved once. Incremental checkpointing implies having a full checkpoint at least once (see Figure 2.2). With forked checkpointing, a clone process is generated for each process. Most operating systems implement process cloning with copy-on-write [LNP94], in which a process and its clone share a memory page until one of them changes this page. This page sharing reduces the memory usage.

Diskless checkpointing can use mirroring and parity methods for redundancy. A process checkpoint can be reconstructed from its neighbor's process checkpoint in the same group and from party information. Redundancy schemes like redundant array of inexpensive disks (RAID) Level 5 [CLG+94, PGK88], Hamming codes [Gib92], Reed-Soloman coding [Pla97] can be used. Reed-Soloman coding achieves maximal failure coverage per checkpoint process, but it has a high CPU overhead [BE09, CSWB08, FL05, Lee03]. Bautista-Gomez et al. hide the latency of Reed-Soloman encoding of checkpoints by using a dedicated MPI process and GPUs [BGTK+11].

2.2.7 Multi-level Checkpointing

An application checkpoint can be stored locally on a computing node. It uses one of the available local storage media, such as, RAM, Flash, SSD or disk. The checkpoint can also be stored remotely to a neighbor node or a network file system. Each checkpoint storage location has different storage bandwidth and has a different level of resilience.

A RAM disk offers more bandwidth than an SSD. Within a computing node,

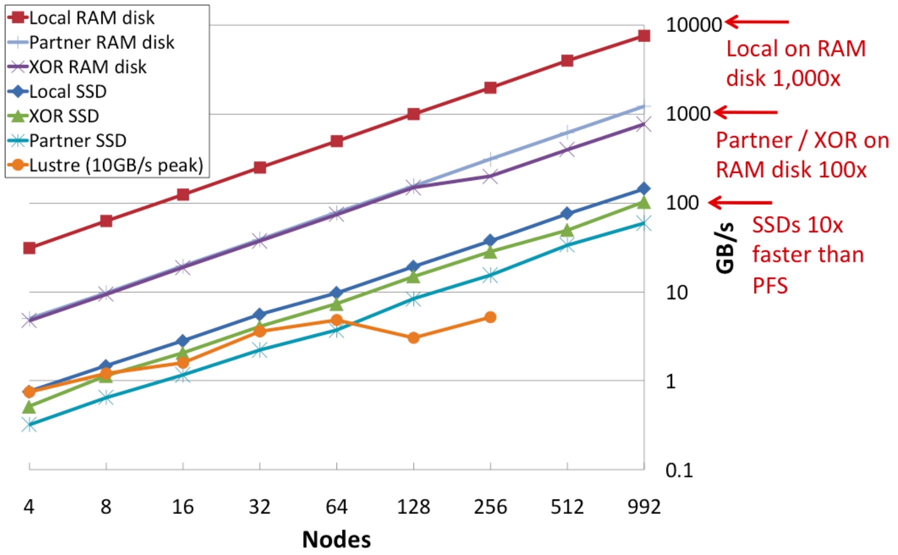


Figure 2.5: Aggregate checkpoint write bandwidth to RAM disk, SSD, and Lustre PFS [MBMS10]. The aggregated write bandwidth on the local RAM disk is 1,000x more than the PFS. The Partner/XOR RAM disk is 100x more than the PFS. Finally, the SSD is 10x more than the PFS.

usually more memory bandwidth is available for local storage than external storage. The parallel file system (PFS) usually have the least bandwidth per core. Its bandwidth is shared between all nodes in the cluster. Applications compete to get a share of the PFS bandwidth.

The available storage bandwidth defines the checkpoint cost. The most expensive storage (concerning time) is the PFS, and the cheapest is the local RAM disk [MMBdS14]. Checkpointing to PFS offers the highest resilient level as it can recover failure of an entire computing node. However, checkpointing to local storage has less resilient.

Figure 2.5 shows the aggregate checkpoint write bandwidth to different storage locations measured by Moody et al. [MBMS10]. In local mode, checkpoints are written to local node storage. In partner mode, checkpoints are written to local node storage and a neighbor node storage. In xor mode, redundant checkpoint parity data is stored to local node storage and a set of neighbor nodes. Lustre PFS delivers a peak bandwidth (in their setting) of 10GB/s. It is observed from Figure 2.5 that partner and xor modes with RAM disk achieve 100x more performance than PFS. Local mode with RAM disk gives 1000x more performance that PFS.

Multi-level checkpointing [Vai94] uses different storage levels with different costs and resiliency. Multi-level checkpointing uses cheap checkpointing levels to handle most common failures and expensive checkpointing levels to handle less common but severe failures. Cheap checkpoints are frequently taken while expensive high resilient checkpoints are taken less often. This increases the efficiency and decreases the load on the parallel file system.

Moody et al. evalute the effectiveness of multi-level checkpointing in large scale HPC systems [MMBdS14, MBMS10]. They implement a system, called Scalable Checkpoint/Restart (SCR) library, that writes checkpoints to storage on the compute nodes utilizing RAM, flash or disk, in addition to the parallel file system. They show that multi-level checkpointing improves efficiency on existing large-scale systems. In particular, they develop low-cost checkpoint schemes that are faster than the parallel file system and effective against 85% of their system failures. This increases the machine efficiency up to 35%, and it reduces the load on the parallel file system by a factor of two.

2.2.8 Asynchronous Checkpointing

A parallel file system (PFS) usually offers the least write bandwidth per core. All nodes in a cluster compete to take a share of this PFS bandwidth. Writing checkpoint files to a PFS is expensive. A single checkpoint can take on the order of tens of minutes [MMBdS14].

Asynchronous checkpointing sends checkpoints to the PFS asynchronously in a way to lower the checkpointing overhead. This is performed by overlapping computation and writing checkpoints to the PFS.' In other words, the application continues execution while checkpointing to the PFS is performed in the background. Nevertheless, the rate of asynchronous sending the checkpoints to the PFS can be tuned to decrease the effective load on the PFS [MMdS12].

Plank et al. use compression with asynchronous checkpointing to lower PFS checkpointing overhead [PL94]. Mohror et al. employ asynchronous checkpointing with multi-level checkpointing [MMdS12]. They enhance Scalable Checkpoint/Restart (SCR) library by using MRNet, a tree-based overlay network library [RAM03], to copy checkpoints from the computing nodes to the parallel file system asynchronously. They show that integrating SCR with MRNeT reduces the time spent in I/O operations. Abbasi et al. show that data staging services moving application output data from compute nodes to dedicated staging or I/O nodes before storing the data to the PFS are used to reduce I/O overheads on applications' total processing times [AWE$^+$09]. A data staging service intends to copy data to an intermediate location so that later the data is copied asynchronously to the Parallel File System (PFS).

Santo et al. extend the work of Abbasi et al. and Mohror et al. to design a non-blocking checkpointing system that combines the benefits of asynchronous checkpointing, multi-level checkpointing and data staging services [SMM$^+$12]. Their system uses the Scalable Checkpoint/Restart (SCR) library to perform multi-checkpoint levels with different cost and resilience.

Figure 2.6 shows the non-blocking checkpoint system of Santo et al. The system has two node sets, computing nodes, and staging nodes. The computing nodes are nodes where applications execute. The staging nodes are nodes that transfer the checkpoints from the computing nodes to the PFS. The staging nodes asynchronously read checkpoint data from compute nodes and write checkpoint data to the PFS. Each staging node handles multiple computing nodes. The staging node reads checkpoints from the computing nodes using Remote Direct Memory Access (RDMA) to minimize the CPU usage on the computing nodes. This is how the system provides the none blocking feature.

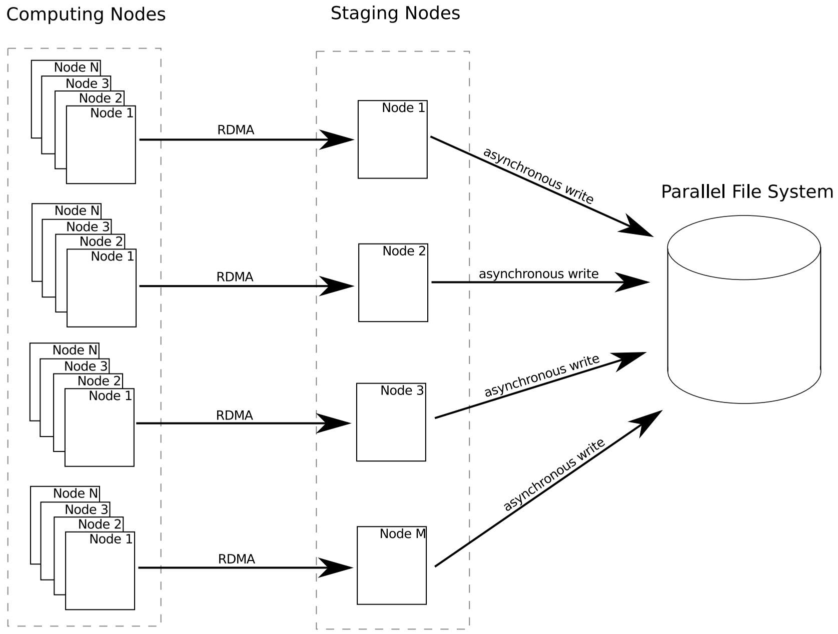


Figure 2.6: Santo et al. non-blocking checkpointing system [SMM$^+$12]. The arrows show the direction of flow of the checkpoints.

2.2.9 Cooperative Checkpointing

Periodic checkpointing is a standard approach for providing fault tolerance for long-running jobs. On a checkpoint, the application execution is interrupted so that it can save its checkpoint to stable storage. With checkpointing more frequently, the application spends most of its time saving its status instead of doing a useful computation. This also dramatically increases the total execution time. CPU time, memory, disk, and parallel file system (PFS) bandwidth are also wasted on checkpoints which are less likely to be used.

Checkpointing less often decreases the benefit of checkpointing; the computation performed by an application from the failure point to the last available checkpoint will be wasted. To avoid both extremes, the checkpoint interval should be carefully tuned. Finding an optimal checkpoint interval has been widely studied for single process applications [PE98, Vai97] and multiprocess parallel applications [PT01, KS97]. The optimal checkpoint interval guarantees the lowest checkpoint rate for a certain rate of failures.

However, periodic checkpointing cannot cope with the dynamic reliability challenges of large scale systems. As a result, cooperative checkpoint is employed [ORS06b, ORS06a]. At runtime, an application requests a checkpoint. The system decides whether to grant or skip the checkpoint based on a dynamic system-wide heuristics. In this way, the application and the system are all part of the decisions regarding when and how checkpoints are performed. Skipping checkpoints which are less likely to be used for recovery improves the overall system performance.

When a periodic checkpoint requested by an application after that the system denies some of these checkpoints, the checkpoint period becomes irregular. So cooperative checkpointing can appear as an irregularity added to the checkpoint interval.

Decisions regarding whether to accept or deny checkpoints are based on heuristics. When good heuristics are chosen, cooperative checkpointing increases the system resilience and decrease checkpoint overhead. The chosen heuristics can depend on the system dynamic changing status, such as, the currently available memory, disk, and PFS bandwidth. The system will skip a checkpoint if less PFS bandwidth is available. In this way, it will avoid having a long checkpointing time.

Also, the chosen heuristics can depend on the expected failure rate. When failures are predicted before they occur, the system will accept to save checkpoints before failures happen. However, when the system is predicted to be stable, the system could skip checkpoints. In this way, the system can adjust the checkpoint period according to the dynamically changing failure probability. Failure prediction can be accurate on real systems [LZS+06, SOR+03]. Figure 2.7 explains the application behavior with checkpoint skipping when failures can be predicted.

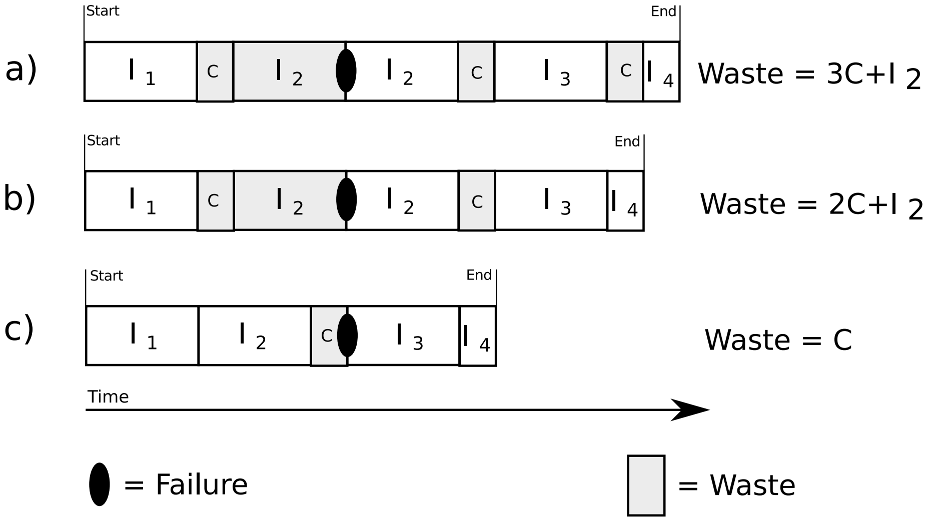


Figure 2.7: Three jobs run with different checkpoints skipping [ORS06b]. Run (a) shows a standard periodic checkpoint, in which all checkpoints are performed. In run(b), the last checkpoint is skipped because the system predicted a low probability of failure, given the short time remaining in the computation. Run (c) shows the ideal case, in which a checkpoint is performed just before failure. With periodic checkpointing in the run (a), the waste is at maximum. With cooperative checkpointing in the run (b) and the run (c), the waste is relatively reduced.

Decisions regarding whether to accept or deny a checkpoint can also be taken based on Quality of Service (QoS) guarantees [ORS+05]. For example, a late starting job can be made to skip checkpoints to reduce its effective runtime. In this way, the job is more likely to meet its deadline.

2.2.10 Checkpointing Filesystem

Parallel applications run across thousand of processors and simultaneously save their checkpoints to a stable storage. In the case of failure, they can rollback to the last available checkpoint. Checkpointing to the parallel file system (PFS) allows the highest level of resilience as it can survive a complete power outage. However, PFS bandwidth is shared between all computing nodes. So checkpointing to the PFS is a potential source of system bottlenecks. Single application checkpoint can last for a tens of minutes [MMBdS14]. Checkpointing has become the driving workload of PFS and the challenges it imposes grow with every new larger supercomputer [KGS+16, MBMS10, EP04].

Usually, different HPC applications have different checkpointing patterns. Some of these patterns lead to very poor storage performance on PFS systems like PanFS, GPFS [VTHB18], and Lustre [QLI+17]. Bent et al. propose a virtual parallel log structure file system (PLFS) that remaps the application checkpointing pattern into one which is optimized for the underlying filesystem [BGG+09]. PLFS acts as an interposition layer inserted into the existing storage stack that rearranges an applications checkpoint pattern into one which is optimized for the underlying filesystem. Bent et al. evaluate PLFS on PanFS, GPFS, and Lustre. They show the PLFS reduce the checkpointing time of their tested applications by an order of magnitude.

Cranor et al. structure PLFS to become a more general platform for transparently translating application I/O patterns [CPG13]. They extend PLFS so that it can be used with different backend storage, for example, cloud storage, such as Hadoop Distributed Filesystem (HDFS), and to solve different workload problems. HDFS is not POSIX-based and does not support multiple concurrent writes to the same file [SKRC10]. However, with the extended PLFS, HPC applications have the ability to concurrently write from multiple computing nodes into a single file stored in HDFS.

Multi-level checkpointing [MMBdS14] is also used to decrease the load on the PFS. It uses the node local storage for storing low overhead, and high frequently checkpoints. Only a few selected checkpoints are stored on the PFS. A node local storage has an advantage that it scales with the number of the used nodes by an application. The more computing nodes are used, the more accumulated storage capacity is available for the application. Unfortunately, not every HPC system provides a local node storage. They have only the local node memory.

Rajachandrasekar et al. develop a user space file system, called CRUISE, which stores data in main memory [RMMP13]. CRUISE is optimized to be used with a multi-level checkpointing library like Scalable Checkpoint/Restart (SCR). In this way, multi-level checkpointing can still be performed on computing nodes which do not have local storage. CRUISE transparently spills data to other storage like local flash memory or the parallel file system. It also supports Remote Direct Memory Access (RDMA), so that it can allow other remote server processes to read files directly from a computer node memory.

2.3 Checkpointing Distributed Parallel Applications

Checkpointing a single process application saves the process status to a stable storage. In a case of failure, rollback-recovery restore the process status from the latest checkpoint. However, for a parallel multi-process application, such as, MPI applications, the checkpoint and restart operations are not that simple. The application has multiple processes running in parallel. The processes are not necessarily sharing the same computing node. They may be physically or logically isolated from each other. The processes communicate with each other using message passing mechanism for example MPI [GL93] (see Figure 2.8). The global checkpoint of a parallel distributed application includes the state of each participating process and the state of the communication channel.

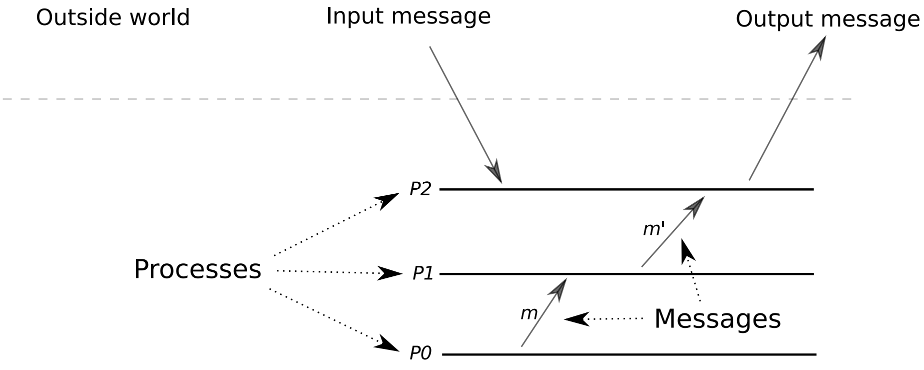


Figure 2.8: A parallel muli-process application [EAWJ02]. Processes communicate with each other using message passing mechanism. They can also send and receive messages from the outside world. The processes can be logically or physically isolated from each other, i. e., not necessarily running on the same computing node.

Just having a checkpoint for every process in this message passing system may not be sufficient to restore the status of the application in a case of failure. In-flight messages should also be restored after roll-back, i. e., restart. If the system fails in restoring the in-flight messages, it tries to restart again but from an older checkpoint. If it is still unsuccessful, the action can be repeated. This is known as rollback propagation [EAWJ02].

For example, if a sender of a message m restarted from a state before sending m, the receiver process must restart back to a state before receiving m. Otherwise, the states of the two processes would be inconsistent, they would show that the message m was received but has never been sent, which is impossible for a normal program execution.

In some cases, cascaded rollback propagation can be relatively long, it can force the application to return to its initial state, i.e., as if it is just starting, losing all work done before failure. This unbounded rollback is referred in literature as the domino effect [KT87, EAWJ02] (see Figure 2.9). Having the domino effect in the rollback process is completely unacceptable. As it makes the checkpoint process useless; all resources spent on checkpointing including CPU time, memory and disk space are wasted in vain.

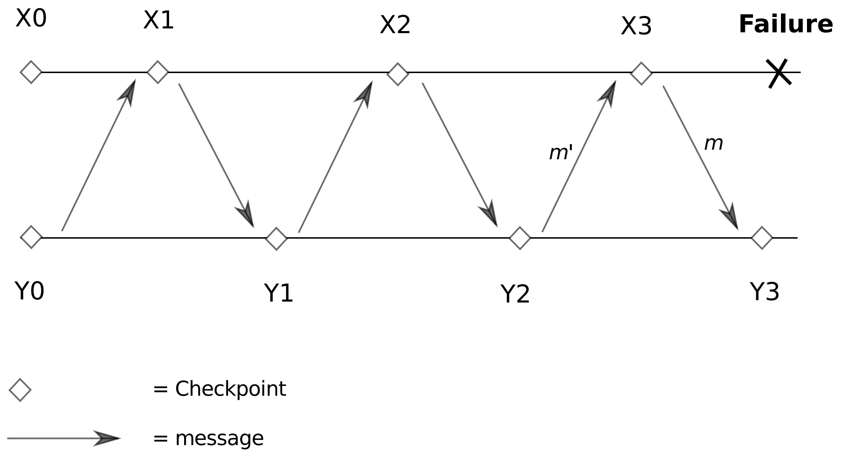


Figure 2.9: The domino effect [KT87, EAWJ02]. For processes X and Y, their earliest checkpoints are ($X0$, $Y0$) and their last available checkpoints are ($X3$, $Y3$). When X process fails, the application tries to restart from ($X3$, $Y3$). However, ($X3$, $Y3$) checkpoints are inconsistent, $Y3$ shows that m message was received, but $X3$ has not sent m yet. As a result of this restarting from ($X3$, $Y3$) fails. The application tries to restart again from earlier checkpoints ($X3$, $Y2$), ($X2$, $Y3$), etc. This is known as rollback propagation. But unfortunately, all these checkpoints are inconsistent. So it is only possible to restart the application from the earliest checkpoint ($X0$, $Y0$). Unbounded cascaded inconsistent checkpoints force the program to return to the beginning in case of failure. If this happens, checkpointing becomes useless. This is what is know as the domino effect.

In a message passing multi-process application, if every process takes the decision regarding when to take its checkpoint by it own independent of all other processes, then the system is vulnerable to the domino effect. This approach is known as uncoordinated checkpointing or independent checkpointing [GC11, EAWJ02, FWL$^+$14].

One way to prevent the domino effect is to perform coordinated checkpointing [EAWJ02, FWL$^+$14, ADY12, CS98]. All processes communicate with each other to find a system-wide consistent checkpointing state, where there is no in-flight messages that cannot be restored. This consistent state limits the rollback propagation.

Communication-induced checkpointing is also used to evade the domino effect [BHMR97, MG09, EAWJ02]. Each process is forced to take a checkpoint based on messages it receives from other processes. Certain communication patterns are chosen such that they guarantee a domino effect free rollback.

In the next sections, we try to understand what a consistent global checkpoint is, and what are factors that make a checkpoint inconsistent. Then, we discuss checkpointing techniques for a message passing multi-process application. The techniques include uncoordinated, coordinated and communication-induced checkpointing. In our discussion, we are not interested in implementation details regarding different message types (synchronous or asynchronous) or different library implementations.

2.3.1 Consistent Global Checkpoint

The global checkpoint of a multi-process message passing application is composed of the states of the participating processes, i. e., local checkpoints, and the state of every communication channel between these processes. The global checkpoint is consistent if every message received is also shown as sent in the state of the sender [Wol10b, EAWJ02]. This occurs in a normal execution of the program.

For example, Figure 2.10a shows a consistent checkpoint while Figure 2.10b shows an inconsistent checkpoint. The arrows represent the messages, and the dotted lines describe the global checkpoint. In Figure 2.10b, the checkpoint of *P2* tells that a message was received. However, the checkpoint of *P1* indicates that the message was never sent. This is inconsistent and should not happen in a normal execution of the program.

Messages that are sent but not yet received may not cause the checkpoint to be inconsistent. For example, message *m'* in Figure 2.10a, the checkpoint of *P0* tells that *m'* was sent but the state of *P1* indicates that *P1* did not try to receive the message. This kind of message is called in-transit message. In a fault tolerant system, the sender *P0* resends this message again [Wol10b]. In-transit messages do not invalidate a global system checkpoint in fault tolerant system.

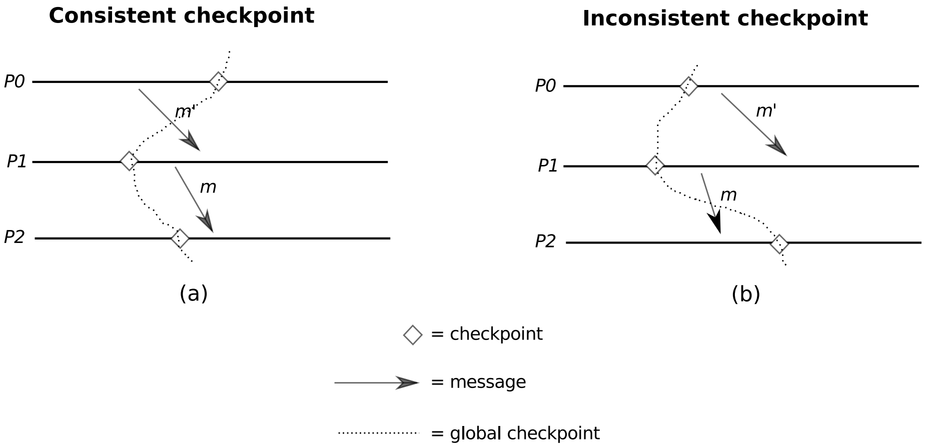


Figure 2.10: Consistent and inconsistent global checkpoint [Wol10b, EAWJ02]. A global checkpoint is consistent if all the checkpoints of all the contributing processes do not record that they have received a message that was never sent. For example in (b) the checkpoint of *P2* shows that it has received the message *m*. However, the checkpoint of *P1* indicates that the message *m* was not sent yet. This is an inconsistency that should not occur in a normal program execution.

The checkpointing protocol of a multi-process application periodically saves the status of each process to stable storage. If a failure happens, the application restarts from the latest available consistent checkpoint. Inconsistent checkpoints are skipped. The latest consistent global checkpoint is called the recovery line [OAFI04, EAWJ02].

The recovery line is used to restore the global status of the complete application. Figure 2.11 shows the recovery line where all checkpoints after the recovery line are inconsistent. In the case of failure, all the work done after the recovery line is lost. For unbounded rollback, i. e., the domino effect, the recovery line is at the beginning of the program.

2.3.2 Uncoordinated Checkpointing

Uncoordinated (or independent) checkpointing of a multi-process application allows every process to decide when to take its checkpoint independent of the other processes [GC11, EAWJ02, FWL$^+$14]. The advantage of this is that every process has the choice in selecting the checkpoint location with a possible minimum checkpoint size which in turn decreases the checkpoint overhead. All the participating processes do not need to do their checkpoints at the same time which reduces the I/O stress on the system. No communication between the processes is required for the checkpointing which decreases the checkpointing overhead.

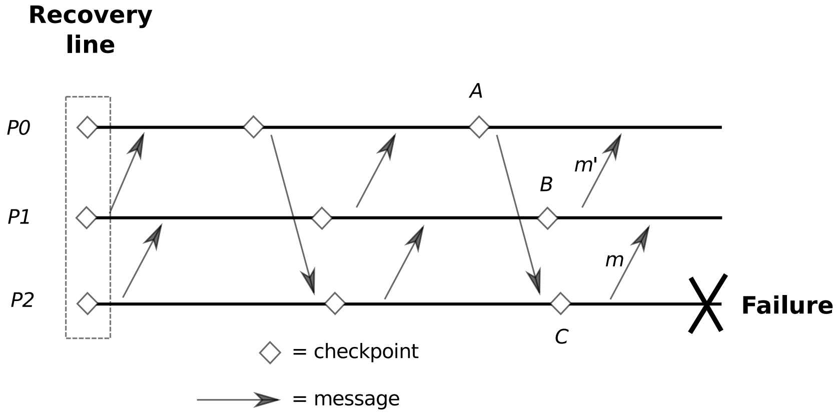


Figure 2.11: The recovery line [EAWJ02]. In failure events, inconsistent checkpoints are skipped until a consistent checkpoint is found. The last recent consistent checkpoint is known as the recovery line.

The disadvantage of uncoordinated checkpointing is that it may skip checkpoints until a consistent global checkpoint, i. e., recovery line is found. This makes it vulnerable to the domino effect. Figure 2.11 shows an application composed of three processes that are allowed to have checkpoints independently without any coordination between them. The diamond figures represent the checkpoints; it is assumed that every process has an initial checkpoint at the beginning of the execution. Suppose that process *P2* failed; it has to rollback to checkpoint *C*. Restarting from *C* will "unsend" message *m*. As a result of this, process *P1* has to rollback to *B* to "unreceive" message *m*. This means that the rollback of process *P2* has propagated to *P1*. This is where the name rollback propagation comes from. *P1* restarting from *B* will unsent message *m'*. This will force *P0* to rollback to *A*. Such cascaded rollback can lead to unbounded rollback, i. e., the domino effect.

During normal application execution, there are dependencies between local checkpoints. These dependencies are due to the message exchange between the processes. The dependencies are recorded so that they are used later to calculate the recovery line [EAWJ02].

Checkpoints that are no longer required can be deleted, as only the recovery line is sufficient to restore the global state of the application. A garbage collection task can be triggered periodically to calculate the new recovery line and delete unnecessary checkpoints.

Uncoordinated checkpointing can be mixed with message logging to avoid the domino effect [FWL$^+$14, GC11]. Message logging can be used to recover a process state as declared by the piecewise deterministic execution model (PWD) [EAWJ02]. The model states that process execution time can be divided into deterministic state intervals. Each interval starts by a non-deterministic event. By logging and replaying the non-deterministic events in their original order, a process can reconstruct its status even if it does not have a recent checkpoint. This recovery approach is useful for applications that interact with the outside world. However, logging messages introduces overhead. Several previous works have been done to reduce this overhead [GC11, RGU$^+$11].

2.3.3 Coordinated Checkpointing

In coordinated checkpointing, all processes of a parallel application communicate with each other to agree on a consistent checkpoint [EAWJ02, FWL$^+$14, ADY12, CS98]. The advantages of this are that always the recent checkpoint is consistent. The application can be restarted from this checkpoint. Calculating the recovery line is simple because the last checkpoint becomes the recovery line. As a result of this, coordinated checkpointing is free from the domino effect. Old checkpoints can be deleted because they are no longer needed.

The disadvantages of coordinating checkpoint are that each contributing process has no longer the privilege of choosing its local optimal checkpoint. All processes have to communicate with each other to find a consistent checkpoint; this requires a bulk of messages which generate a time overhead and stress the communication channel. All participating processes save their checkpoints within a short time window which stress the system I/O. If any participating process fails, all processes are forced to restart. Global processes restart is a waste of energy and time [GC11].

The simplest approach for performing coordinating checkpointing is by blocking interprocess communications and drain all in-flight messages by sending a broadcast checkpoint request. Once a checkpoint request is received, each process stops sending any more messages then starts performing its local checkpoint [KT87,KP93,CHL$^+$06]. When all processes are done, a broadcast message is dispatched to all process so that they can continue execution. The disadvantage of this is mainly the wasted time in blocking.

One way to avoid blocking is to make the receiver process responsible for checkpoint consistency. This is explained by the example in Figure 2.12. In Figure 2.12a, inconsistency occurs when a checkpoint request was sent to *P0* and *P1*; however, the request has arrived at different time. *P0* has a checkpoint before sending *m*, and *P1* has a checkpoint after receiving *m*. Now the status of *P1* shows that m was received; however, the status of *P0* shows that *m* was never sent.

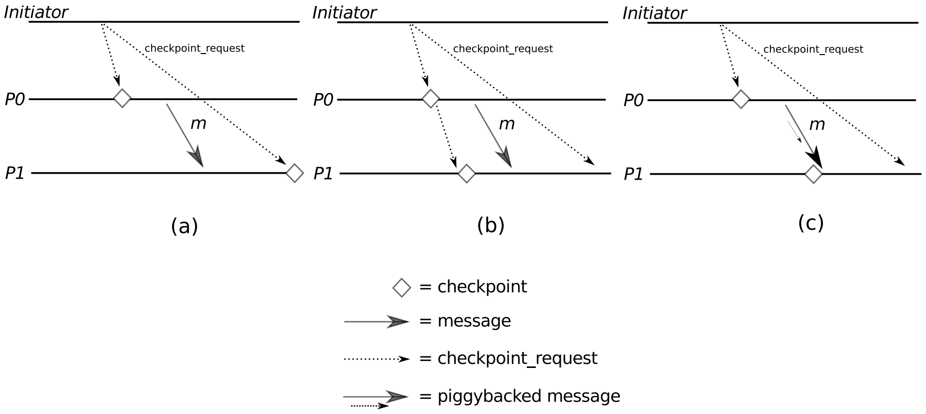


Figure 2.12: Nonblocking coordinating checkpointing [EAWJ02]. (a) An inconsistent checkpoint. (b) *P0* is forced to send a checkpoint request before sending *m*. *P1* is forced to have a checkpoint upon the first received checkpoint request. (c) The checkpoint request is piggybacked with the message. The dotted short arrow represents the piggybacked checkpoint request.

For a FIFO channel, this problem can be solved by forcing every sender process *P0* to send a checkpoint request before sending its message *m* and forcing every receiver process to take a checkpoint once it receives the first checkpoint request [CL85, CHL$^+$06] (see Figure 2.12b). If the system does not guarantee a FIFO channel, i. e., in Figure 2.12b, for process *P1*, the checkpoint request may not arrive before *m*. The checkpoint request can be piggybacked with the message [LY87] (see Figure 2.12c). The receiver *P1* has the possibility to read the piggybacked label before processing the message *m*. If the piggybacked information contains a checkpoint request, the receiver *P1* is forced to have a checkpoint before processing message *m*.

2.3.4 Communication-induced Checkpointing

Communication-induced checkpointing combines coordinated and uncoordinated checkpointing to get the benefit of both. It avoids the domino effect and still allows processes to have freedom in choosing their checkpoints [MG09, EAWJ02, BHMR97]. It does not require special coordination messages to be sent between the process. However, it enforces system-wide constraints on the communications and checkpoints. These constraints guarantee a consistent global checkpoint. In other words, they guarantee progression of the recovery line. Participating processes are occasionally forced to have checkpoints due to these constraints.

Communication-induced checkpoint achieves coordination between the system processes by piggybacking control information with messages. A receiver process can read the control information without opening the received message. Based on the control information, the receiver process might be forced to have a checkpoint according to the system-wide constraints.

Table 2.1: TSUBAME2.0 failure categories [SMM+12]. Single node failure category has the highest failure rate.

Category	# of affected nodes	Failure points	Failure rate (failure/sec)	MTBF
5	1408	PFS, Core switch	$0.1778e-6$	67.10 days
4	32	Rack	$0.1332e-6$	86.90 days
3	16	Edge switch	$0.6665e-6$	17.37 days
2	4	Power Supply Unit	$0.3999e-6$	28.94 days
1	1	Compute nodes	$0.1757e-4$	15.8 hours

2.4 Checkpointing Applications

In this section, we provide an overview of checkpointing applications which include fault-tolerance, playback debugging and migration. We also discuss failure rates in HPC systems.

2.4.1 Fault-tolerance

The application saves its checkpoint periodically to a stable non-volatile storage.

In case any of the application's processes fail, all of its processes may roll back to the last checkpoint and restart from there [EAWJ02]. Checkpointing is vital for long running applications. Otherwise, in the case of failure, the application needs to restart from the beginning.

Failure rate

Understanding how often failures occur helps in choosing a suitable checkpointing scheme. The failure rate of an HPC system increases with its size. For current petascale systems, the Mean Time Between Failures (MTBF) is in the order of hours [SG07b, SMM+12]. However, it is expected that it will further decrease for exascale systems [DBM+11, BBC+08, GCR+07].

A study of the failure history of TSUBAME2.0 [Tsu], a petascale system ranked 5th on the Top500 [Top] list in November 2011, is shown in Table 2.1. Santo et al. have done this study between November 1, 2010, and April 6, 2012 [SMM+12]. The study classifies failures into categories or levels according to the number of affected computing nodes. The study shows that most failures are in category 1, which means that they affect only a single computing node. Its rate is two order of magnitude larger than the failure rates of the other categories. This observation is the core of the design of multi-level checkpointing discussed in Section 2.2.7.

Furthermore, the study investigates the failure rate of components within a single node, see Figure 2.13. The study shows that approximately half of compute node failures arise from GPUs (Nvidia Tesla M2050). As more applications use GPUs, the overall system temperature increases. This results in a high failure rate of GPUs. Our contribution in Chapter 3 can be used to recover GPU application in the case of failures.

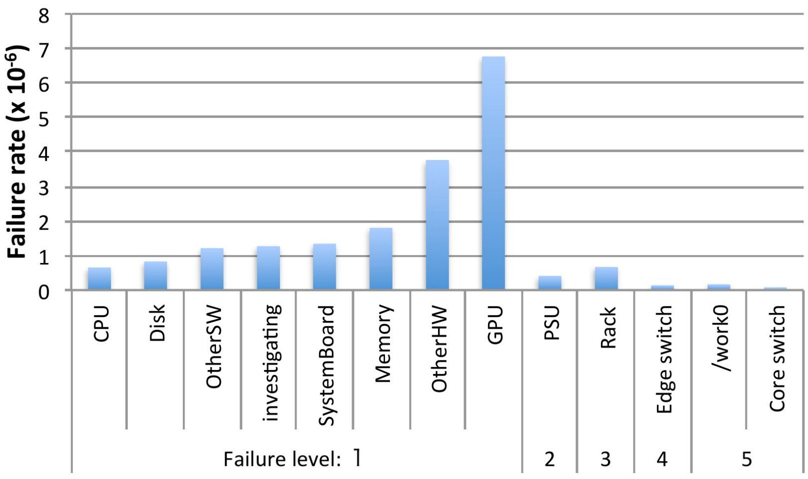


Figure 2.13: The failure history of TSUBAME2.0 HPC [SMM^{+}12]. Single node failures are most common (Failure level 1). GPU failures have the highest failure rate.

2.4.2 Playback Debugging

Programs usually have programming errors. Programmers probably spend more time in debugging than in coding their programs. Due to bugs, program errors can appear during the runtime of a program which can last from hours to days. Some of these errors can cause complete failures. With the help of checkpointing, a programmer does not need to restart a program from the beginning to regenerate the same error again. Instead, the program can be restarted from nearest checkpoint to that error. The programmer also has the possibility to attach a debugger to his program when restarting from the checkpoint. This way, checkpointing facilitates the debugging process and decreases its time.

However, in some cases, this approach does not work. Some errors that appear during a program runtime and cause program failure might not appear when restarting this program from the last checkpoint. This is due to the change in the environment status at the time of restarting from the checkpoint [WHV^{+}95].

Listing 2.3: Heap memory allocation failure. This type of errors might not be regenerated upon restarting from a checkpoint.

```
ptr = malloc(size);
if(ptr == NULL){
printf(”ErrorAllocatingMemory\n”);
return EXIT_WITH_ERROR;
}
.... //Use ptr
```

A real life example of these types of error is shown in Listing 2.3. It shows a program segment that is quite common in Linux programs which dynamically allocate memory from the heap. When this program fails to allocate memory, it will show an error message, then exit. This kind of software exit is undesirable for a long running application. With the help of checkpointing, the program can be restarted from its last checkpoint. On restarting, this error might not appear again as the environment status might have changed, i. e. memory might become available on the system again and can be allocated.

2.5 Migration

Application migration is the operation of moving computations from one computing resource to another. It allows a clean separation between hardware and software. It is widely used for fault tolerance [WMES12], hardware maintenance and workload balancing between computing nodes. Application migration techniques can be divided into four groups: process-level migration, kernel migration, virtual machine migration and container-based migration [PGL$^+$14].

2.5.1 Process-level Migration

Process level migration is based on checkpointing. To migrate a running application from a source node to a destination node, a checkpoint is taken on the source node then this checkpoint is copied to the destination node. Finally, the application is restarted from that checkpoint on the destination node [Rom02]. Any checkpointing technique discussed in Section 2.1 can be used.

A successful application migration is only possible when all resources that are allocated by the application on the source node, i. e. the residual dependencies, are provided by the destination node as well [MDP$^+$00]. Unfortunately, process-level migration cannot always guaranty to provide resources like open files and communication channels on the destination node. For example, an open file descriptor on the source node will not be valid on the destination node. The file descriptor could be closed before migration, which implies a non-transparent migration from the application's perspective.

2.5.2 Kernel Migration

Accelerators like GPUs, DSPs, and FPGAs can be used to boost performance by exploring parallelism within an application. For example, GPUs are massively parallel devices that combine many-core multi-tasking with vectorization. NVIDIA CUDA GPUs are built of a scalable array of multiprocessors [Cla07]. Each multiprocessor has a Single Instruction Multiple Data (SIMD) architecture. In this architecture, each processor of the multiprocessor executes the same instruction but operates on different data. They share the same instruction fetch unit (see Figure 2.14).

Using profiling, the computationally expensive part (a kernel) of an application is determined. If possible, this part is parallelized then migrated to be executed on the GPU. We call this operation as kernel migration in this thesis.

Accelerators sometimes have an isolated memory from their host systems. Migrating a kernel to an accelerator implies also migrating the data objects required for the execution of this kernel by time-consuming copy operations. NVIDIA solves this problem by having a unified memory between the host system and the accelerator. This solution was implemented in NVIDIA "Pascal" GPUs. In our contribution in Chapter 3, we propose another solution that

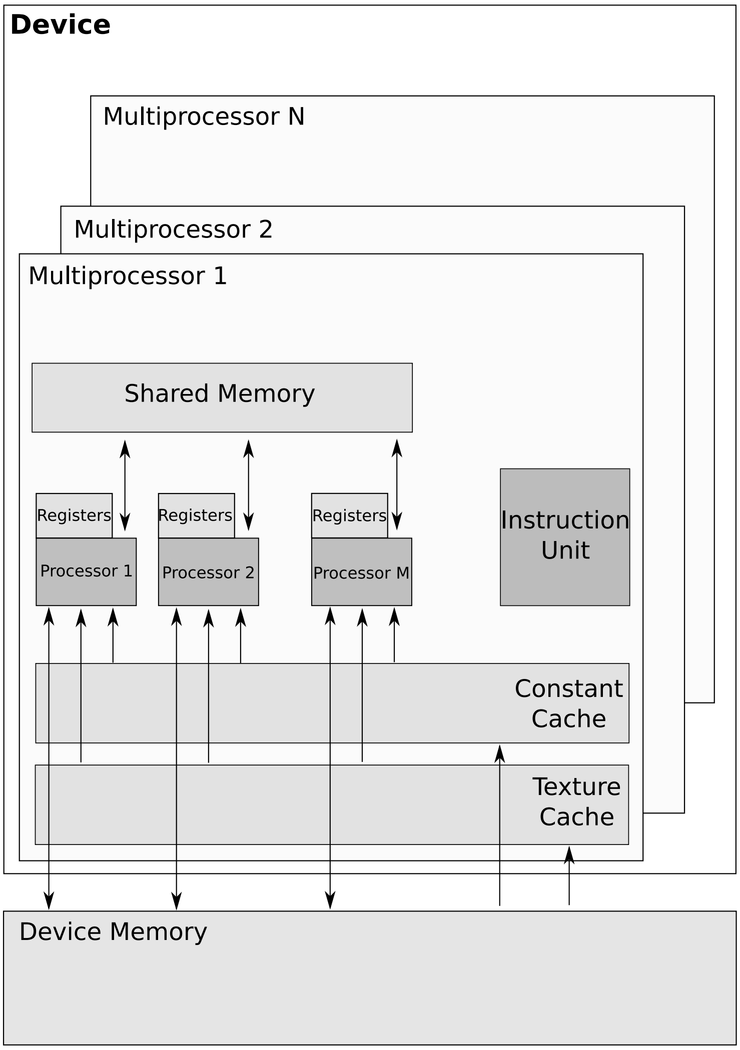


Figure 2.14: SIMD multiprocessors architecture of NVIDIA GPUs. [Cla07]. The GPU has many multiprocessors. Each multiprocessor has many processors that execute the same instruction on different data (SIMD). Each process has its registers. The processors share the instruction unit, constant cache, texture cache, and shared memory within a multiprocessor. All multiprocessors within the GPU share the device memory.

automates determining, serializing and copying data objects required for a kernel execution on an accelerator.

Instead of manual kernel migration, source-to-source compilers, such as, hiCUDA [HA09], CAPS HMPP [DBB07] and PGI compiler [Wol10a] can be employed to transfer serial code into parallel architecture specific code for the accelerator. These tools use preprocessor directive-based approach, i. e. pragma in which an application programmer provides some hints. In these hints, he defines parallel regions including loops that should be migrated to the accelerators. He also defines data objects that should be relocated to the accelerator memory.

The success of this preprocessor directive-based approach leads to the definition of an Industrial standard for GPU directive-based programming called OpenACC [Far16]. Its API is inspired by OpenMP [DM98]. In this standard, still, parallel loops are identified by preprocessor pragma. Listing 2.4 shows a sample OpenACC code. OpenACC offers a memory managed mode in which data objects required for a kernel execution are fetched on demand from the host memory to accelerator memory [Far16]. However, this mode results in performance degradation. If memory managed mode is disable, these object should be manually managed by the programmer using OpenACC data clauses like copyin(), copyout(), copy(), create(), delete() and present().

The advantages of using accelerators which include performance enhancement and energy saving [SDG+16b,MV15], make them attractive to incorporate in the system scheduler. In other words, the system scheduler treats them like an ordinary CPU core and assign jobs to them. The scheduler aims to efficiently use all the resources within a compute node according to predefined strategies and the current status of the runtime environment.

Beisel et al. [BWPB11] have developed a resource-aware scheduler that incorporates accelerators in the scheduling process. However, this schedule has a static scheduling strategy. A similar scheduler was provided by Süß et al. called VarSched [SDG+16b]. In contrast to Beisel et al. scheduler, VarSched supports multiple scheduling strategies, dynamic switching between strategies at runtime according to the system requirements and migration of kernels between resources at runtime.

Listing 2.4: Jacobi iteration with OpenACC [Far16]. An application programmer uses pragma directives to identify loops that need to be parallelized (in Lines 4 and 12) and to mark data objects that need to be copied or allocated to the accelerator memory (in Line 1). The pragma directives are not intended for a certain accelerator. The compiler uses the pragma directives to port the marked code regions to a particular accelerator. A non-parallelizing compiler would simply ignore the pragma directives. In this way, the code becomes portable.

```
  #pragma acc data copy(A), create(Anew)
2 while ( err > tol && iter < iter_max ) {
   err=0.0;
4 #pragma acc parallel loop reduction(max:err)
   for( int j = 1; j < n-1; j++) {
6   for(int i = 1; i < m-1; i++) {
     Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] + A[j-1][i]
8     + A[j+1][i]);
```

```
      err = max(err, abs(Anew[j][i] - A[j][i]));
      }
  }
  #pragma acc parallel loop
  for( int j = 1; j < n-1; j++) {
    for( int i = 1; i < m-1; i++ ) {
      A[j][i] = Anew[j][i];
    }
  }
  iter++;
}
```

2.5.3 Virtual machine Migration

Virtual machine migration is an alternative to process level migration, which reduces the problem of residual dependencies [CFH+05]. Resources like open files and virtual I/Os within a virtual machine are still valid after migration. For a successful migration, the memory state of the virtual machine needs to be copied from the source to the destination node, and the Virtual Machine Monitor (VMM) on the destination node needs to provide the same hardware configuration used by the virtual machine on the source node. A VMM is a piece of software that is responsible for housekeeping virtual machines. Furthermore, virtual machine migration supports moving a running application between computing nodes without stopping it, this is known as live migration.

KVM is an open source virtualization solution integrated into the Linux kernel [KKL+07]. It provides full virtualization on x86 hardware depending on the VT-x or AMD-V hardware extensions [UNR+05, Vir05]. A virtual machine is started as an ordinary Linux process running under the host system. If the virtual machine configuration has more than one virtual core, a thread is created per each virtual core, so that the threads can be scheduled separately on the host system.

Figure 2.15a shows a block diagram of KVM virtualization framework. Each virtual machine has its own operating system. The host and the guest operating system do not need to be the same.

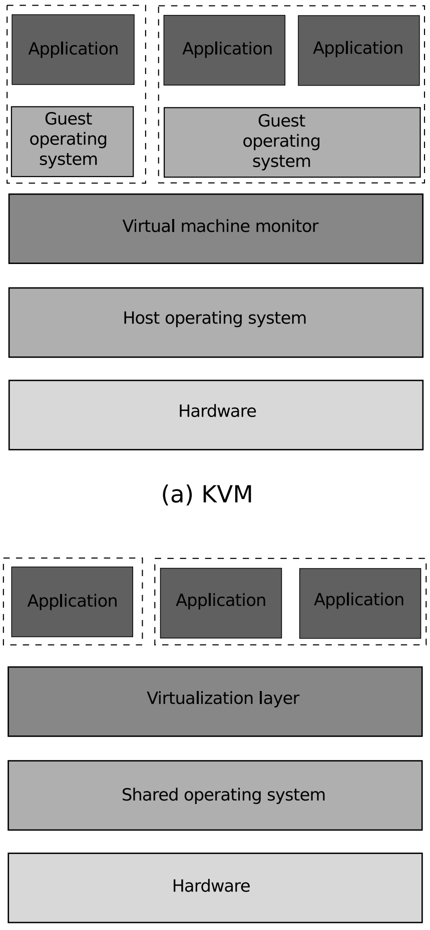
2.5.4 Container-based Migration

Container-based virtualization is a lightweight alternative to a virtual machine. It uses the host operating system to manage what is called a virtual container. A virtual container provides a full virtualization similar to a virtual machine, but it guarantees a better utilization and less overhead. All containers, running on a host node, share execution on the same host operating system, they do not have their own operating system like in the case of virtual machines. Eliminating the redundant operating systems provides a better utilization of the host node. However, this comes at a cost. It is not possible to run different operating systems on the same host, and crashes on the host operating system would halt all the containers running under this system.

Linux Containers (LXC), OpenVZ [RH11], and Docker [Mer14] are common container-based virtualization solutions. Docker has rapidly become a standard tool for managing containers. It is a lightweight virtualization solution that combines Linux containers with an overlay filesystem. It provides utilities for building and packing applications into a portable environment, i. e., container image.

Regola et al. study OpenVZ and show that container-based virtualization solution offers a near-native CPU and I/O performance [RD10]. Figure 2.15b shows container-based virtualization with LXC. Applications are running inside containers which share execution on the same host operation system.

Several works have been performed to compare container-based virtualization and virtual machine based virtualization. Morabito et al. have performed a detailed performance comparison between virtual machine virtualization represented in KVM and container-based virtualization expressed in LXC and Docker [MKK15]. Their results show that containers achieve better performance. A similar performance comparison evaluation was performed by Felter et al. [FFRR15]. However, they are only concerned about KVM and Docker in their comparison. They evaluate KVM and Docker performance using Linpack, Stream, RandomAccess, nuttcp, netperf, and MySQL. Their results correspond with the results from Morabito et al. that Docker outperforms KVM.



(a) KVM

(b) LXC

Figure 2.15: KVM and LXC virtualization solutions. For KVM each virtual machine has its own operating system, while for LXC all containers share execution on the host operating system. Removing the redundant operation system in the case of LXC improves the performance. This is confirmed by the performance comparison study performed by Morabito et al. [MKK15].

Chapter 3

Compiler Driven Automatic Kernel Context Migration for Heterogeneous Computing

Modern computer systems provide different heterogeneous resources, like GPUs, DSPs or FPGAs. Applications can profit significantly from these resources to accelerate computations or save energy. These resources have either isolated or unified memory spaces. While executing applications on these resources, if the memory spaces are separated from each other, data must be moved from one resource to another one manually. In the case where the memory spaces are unified, data is moved automatically but often for the price of a significant performance penalties. Besides these issues, a target specific code has to be created for the different resources. Different tools exist that automatically perform this task [ACE$^+$12, BHRS08, VCJC$^+$13]. These tools take parts of the code, so-called kernels, which have been written for a common CPU as input, and produce target specific kernels that exploit the advantages of their target. However, these tools restrict the programmers to use only primitive data types or simple data structures like arrays in their kernels. When switching between resources, it is in the responsibility of the programmer to serialize the data objects used in the kernel and to copy them to or from the resource's memory. Typically, the programmer writes his own serializing function or uses existing serialization libraries [SLL01, Meh08, KF10]. Unfortunately, these approaches require code modifications and the programmer requires knowledge of the used data structure format. The programmer might need to modify the used data structures or he might be restricted to particular types [AC01].

These issues show that there is a need for a tool that is able to automatically extract the original kernel data objects, serialize them and migrate them between target resources without the programmer's intervention. Kernel migration from the CPU to another target resource allows exploiting all resource advantages.

We refer to data objects used by a kernel as kernel context. A kernel context can contain complex data structures based on primitive data types as well as externally predefined types and data structures in a supplied library like the STL.

In this chapter, we present the tool collection *ConSerner* (Context Serializer) that automatically identifies, gathers, and serializes the context of an original CPU kernel and migrates it to a target resource's memory where a target specific kernel is executed with this data. In this way, the target resource can be used concurrently with the original CPU. This is all done transparently to the programmer. Complex data structures (e.g., n-dimensional arrays, lists, trees, graphs) can be used without the need for manual modification of the program code by a programmer. Predefined data structures in external libraries (e.g., the STL's vector) can also be used as long as the source code of these libraries is available, because our tool get data type information from the source code.

Our tool collection is intended to provide help for the programmer. As it saves the programmer the burden of writing error-prone time-consuming serialization codes manually. In this way, it facilitates the use of heterogenous resources. Migrating a kernel context with its context to a heterogeneous resource exploits the advantage of this resource by providing a code acceleration and saving energy.

Our tool collection provides runtime libraries and a compiler plugin that is partially embedded inside the LLVM compiler framework. In order to detect code regions where data is allocated in the heap memory and where its type is casted, we use the *static compiler analysis* provided by the LLVM compiler framework. We call these regions *actors*.

The original program code is instrumented by injecting code that is responsible for monitoring the actors' behavior at runtime. All memory allocations are recorded including their data size and related type information.

A pointer inside a data structure can point to another memory allocation. In this way, a relation between these two memory allocations is established. The data type information provides us with information about pointers and their locations in the allocated memory. Using this information, pointers are followed to find relations between the allocated memory regions.

The relation between the original kernel's arguments and the allocated memory is explored to identify the required memory regions for the kernel execution. Then, the identified memory regions are gathered and serialized in a single array. The pointer locations in that array are preserved so that they are updated for the target address space. Later the serialized data is migrated to the target's memory and the target kernel is executed. The results are copied back to the CPU memory, deserialized and copied back to the original memory locations.

The main contributions of our tool collection are:

- We show how the LLVM compiler's static analysis can be used to identify regions in the code where memory is allocated and type casted.
- Our tool collection introduces code instrumentation to record the runtime behavior of code regions to provide information about all allocated heap memory regions with their type attribute at runtime.
- Our tool collection automates the identification and serialization of kernel context.

The remainder of this chapter is structured as follows: Related work is discussed in Section 3.1. Section 3.3 describes the structure of our tool collection. Finally, in Section 3.4, we evaluate our tool collection.

3.1 Related Work

Finding kernel contexts is a special type of checkpointing where the checkpoint is generated for a function instead of a complete application. Checkpointing techniques to store the status of applications at a certain point in time have been used extensively to provide support for software fault-tolerance [YG07], playback debugging and process migration [WHV^{+}95]. Takizawa et al. introduced checkpointing for heterogeneous systems [TSKK09]. They presented a tool named CheCUDA, which is able to save the system status and targets NVidia CUDA applications.

CheCUDA is implemented as a plugin in Berkley Lab Checkpoint / Restart for LINUX (BLCR) [Due03]. BLCR does not provide checkpointing for CUDA applications. CheCUDA overcomes this limitation by providing wrapper functions that monitor the CUDA APIs that do the memory allocations, copy data to and from the GPU. During a checkpoint, CheCUDA copies all the GPU memory data to the host memory and destroys the CUDA context. Then, BLCR saves the application's status. On restarting from a checkpoint, BLCR restores the application's status and CheCUDA recreates the CUDA context. In our case, we checkpoint only the data required for the target kernel execution instead of the data required for the complete application execution.

Pourghassemi et al. propose a scalable checkpoint/restart scheme, called cudaCR for long-running kernels on NVIDIA GPUs [Pou17]. Their scheme is able to capture GPU status inside a kernel and roll back to a previous state within the same kernel without destroying the CUDA context, unlike CheCUDA approach.

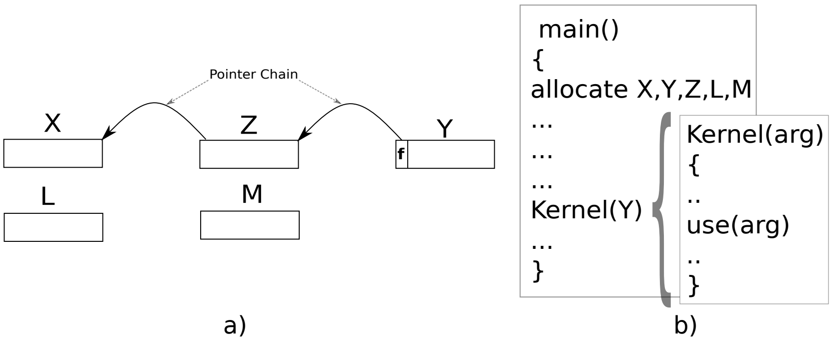
Heterogeneous checkpointing was introduced by Karablieh et al. [KB02]. They proposed a checkpointing technique for multi-threaded applications that use POSIX threads. However, the heterogeneity here is for the different platforms that use this type of threads. That way, they can save the status of the application on one platform and start it on another one. Checkpointing for fault-tolerance in CPU-GPU hybrid systems was introduced by Xu et al. [XLTL10]. They offered an application-level checkpointing tool called "HiAL-Ckpt". This tool requires the user to provide information about the data objects that should be saved at a checkpoint.

Determining the data size of a kernel context is related to program safe execution problem (e.g, array bounds checking, loads and stores only access valid memory objects, etc.). Flater et al. proposed an extended pointer representation (fat pointers) to record information about the intended reference with each pointer [FYP93]. However, the extended pointer representation is incompatible with external unchecked code, e.g., precompiled libraries. Another approach introduced a map, which stores metadata associated with each pointer [DA06]. This approach does not eliminate the compatibility problem of fat pointers. The possibility to modify an observed pointer by an external library requires additional efforts to keep the map up-to-date. Jones et al. proposed to store the address ranges of living objects [JK97] and to ensure that the intermediate pointer arithmetic never crosses from the original object over to another valid object. The address ranges are stored in a global table, which is organized as a *splay tree*. The splay tree is binary search tree that provides a fast access time to recently accessed elements. The table is searched for the intended reference before any pointer arithmetic is performed. This approach removes the incompatibilities introduced by associating metadata with pointers, but it suffers from a high negative performance impact caused by searching the map and keeping it up-to-date. This approach is related to ours, since both approaches keep records of the memory. However, while Jones et al. use the map for boundary checking for each memory access, we just use it to identify the objects' sizes.

Dhurjati et al. proposed a collection of techniques [DA06] to overcome the overhead in Jones et al.'s approach. Their techniques exploit a fine-grained partitioning of memory called *Automatic Pool Allocation*. They were able to bring the average overhead of the runtime checks down to only 12 % for the set of benchmarks that they have evaluated. Later this work was used in the SAFEcode compiler [DKA06]. The commercially available *Purify tool* [HJ91] processes the binary representation of the software. Each memory access instruction is modified to maintain a bit map of valid storage regions and to record if each byte has been initialized. Accesses to unallocated or uninitialized memory are reported as errors. While the Purify tool uses its records to debug memory, to find errors, to detect buffer overflow and improper freeing of memory, we use the records of the valid stored memory regions to know the data size at runtime.

Serialization is the process of translating data structures or objects into a format that can be stored so that they can be restored again in a different environment. The boost library [SLL01] provides a popular serialization interface. Using boost's serial interface requires code modifications and prior knowledge of the used data structures. Protocol buffer [KF10] is a serializing tool that requires the user to explicitly provide the layout of the used data structures. Attardi et al. provided template classes with meta-information, such as storage attributes or index properties of fields [AC01]. A programmer can extend these classes to define objects that can be stored or searched in a database table. This approach forces the programmer to use the predefined templates. ODB (C++) is a compiler based serializing solution that requires the programmer to insert custom pragma directives in the application C++ header files [Meh08]. These directives describe the data objects' layouts, so that they can be serialized and mapped to a relation database. Ser++ [CV13] is a serialization code generator tool that relies on static compiler analysis. It traces pointers and identifies statements in which properties regarding the serialization of pointer attributes can be extracted. Ser++ finds relations between classes and automatically generates the serializing function for these classes. This tool requires the programmer to specify the serializable classes through a special annotation in an external file.

OpenACC is a GPU directive-based programming in which the programmer uses preprocessor pragma to mark parallel loops that should be executed on the GPU [Far16]. OpenACC offers a memory managed mode in which data objects required for a kernel execution are fetched on demand from the host memory to the GPU memory. However, this mode results in performance degradation. If memory managed mode is disable, these object should be manually managed by the programmer using OpenACC data clauses like copyin(), copyout(), copy(),



```
main()
{
allocate X,Y,Z,L,M
...
...
...
Kernel(Y)
...
}
```

```
Kernel(arg)
{
..
use(arg)
..
}
```

Figure 3.1: Objects reachability. Only `X`, `Y`, and `Z` can be accessed by `Kernel()`.

create(), delete() and present().

Most of these tools require code modifications by a programmer to specify how the data has to be serialized. Our tool collection does not require any intervention from a programmer for serialization. The programmer needs to choose the set of kernels that he would like to migrate to the target resource. We analyze the program memory to find relations between the allocated memory regions and to gather the required memory regions for a kernel execution automatically while introducing small overhead.

3.2 Kernel Context

An object is said to be ”live” if it is going to be read or written at some time in the future execution of a program. Unfortunately, liveness is an undecidable property of a program, i. e. there is no way to decide for an arbitrary program during its execution whether it is going to access a particular object or not. The undecidability of liveness is a consequence of the halting problem [JHM16]. If a program holds a pointer to an object this does not mean that it is going to access that object.

For conserner, we are only interested in a kernel context, i.e live objects that are going to be accessed during execution of that kernel, because other objects are not necessary and transferring them costs resources. We approximate liveness by a property that is decidable: pointer reachability [JHM16]. An object `X` is reachable from an object `Y` if `X` can be reached by following a chain of pointers starting from some fields `f` in `Y` (see Figure 3.1 a). By applying pointer reachability, we find out that the kernel contest is formed from all objects that are reachable from the kernel arguments' values by chains of pointers.

An object that is unreachable in the program memory, not pointed to by any kernel argument's value, can never be accessed by the kernel. Conversely, any object reachable from the kernel arguments' values may be accessed by the kernel. Unreachable objects can safely be discarded from the kernel context. Reachable objects are included in the kernel context, even though that they might not get accessed during kernel execution.

Figure 3.1 b shows a sample program which allocates five objects `X`, `Y`, `Z`, `L`, and `M`. The program calls `kernel()` passing `Y` as an argument. The objects

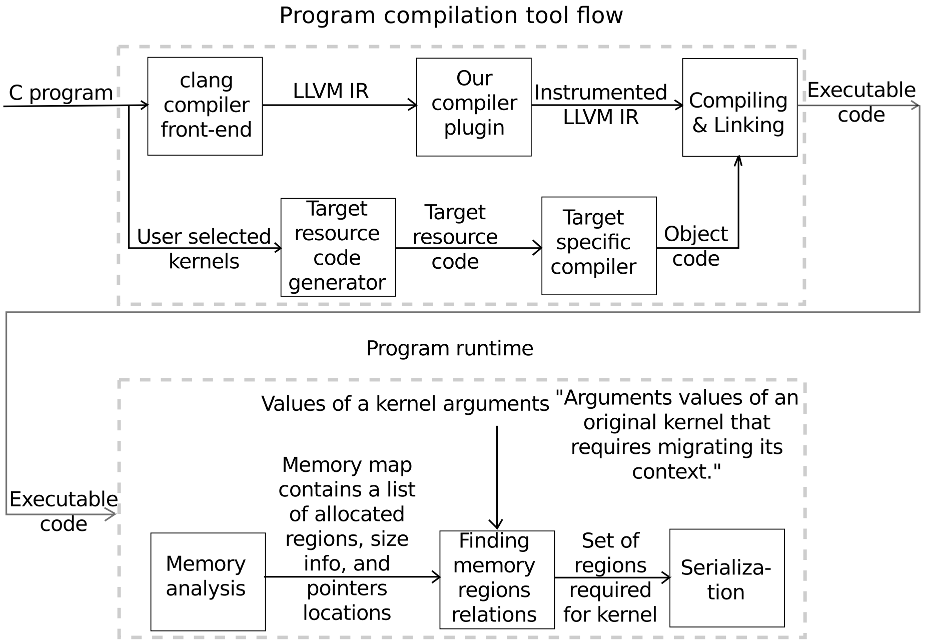

Program compilation tool flow
C program
clang compiler front-end
LLVM IR
Our compiler plugin
Instrumented LLVM IR
Compiling & Linking
Executable code
User selected kernels
Target resource code generator
Target resource code
Target specific compiler
Object code
Program runtime
Values of a kernel arguments
"Arguments values of an original kernel that requires migrating its context."
Executable code
Memory analysis
Memory map contains a list of allocated regions, size info, and pointers locations
Finding memory regions relations
Set of regions required for kernel
Serializa-tion

Figure 3.2: The structure of our tool.

layout in figure 3.1 a shows that only `X` and `Z` are reachable from `Y`. Then `X`, `Y`, and `Z` are the kernel context of `Kernel()`. `L` and `M` will never be accessed by `kernel()` because they are unreachable.

3.3 Tool Flow

This section describes the structure of our tool collection. It relies on two phases: one at program compilation and one at program execution (see Figure 3.2).

3.3.1 Program Compilation Tool flow

In the beginning of the compiling phase, the program C code is analyzed, instrumented, and finally compiled to generate an executable program. Our compiler tool collection uses the Clang compiler [Llva] as front-end to generate an *LLVM intermediate representation* (IR) for the program. The IR of a program is the input for our developed compiler plugin that outputs an instrumented version of the IR. Our plugin is implemented as a compiler pass [Llvc] inside the LLVM framework.

A programmer chooses a set of CPU kernels in his application and uses code generator tools to generate target specific kernels. Later these kernels are compiled and linked to the output of our plugin to produce an executable code. Due to this, the chosen kernels can be migrated to run on the target resource.

Code analysis and instrumentation

Our compiler plugin searches the program LLVM IR for memory allocation functions like `malloc`, `new`. These functions return pointers to allocated memory regions. Our plugin determines where these returned pointers are casted to know the assigned type to the allocated memory regions (see Listing 3.1). Instrumentation functions are injected in the code to record the returned pointers from the allocation functions, the allocation size, and the expected casted data type. These instrumentation functions record information so that it can be used at program execution.

Listing 3.1: The sample LLVM IR code shows the memory allocation function *new*. It returns a pointer *%call*. *%call* is casted later to *%struct.TestStruct*. *%struct.TestStruct* type is composed of two elements a float pointer and a float. An instrumentation function is inserted in this code records *%call* (the address of the allocated region), *i*32 8 (the size of the allocated regions), *%struct.TestStruct* (the region data type). Refer to the LLVM instruction documentation [Llvb] for more info.

```
%struct.TestStruct = type { float*, float }
.
.
%call = call noalias i8*
          @operator new(unsigned int)(i32 8)
.
.
%1 = bitcast i8* %call to %struct.TestStruct*
```

Similarly, our compiler plugin searches the program's IR for memory deallocation functions, like `free` or `delete`, and injects an instrumentation function to record pointers to the freed memory regions.

The limitation of our tool is that it requires the availability of the original source code. It relies on the provided data type information at compilation time. Our tool does not cover situations when an allocated memory region can have more than one type, e.g., a *C union*. Also our tool does not cover situations when data type information is only available at program execution time, for example, if it depends on user input. The current implementation of the tool limits the allocated memory regions to a single casting during the execution of the program.

3.3.2 Program Runtime

During a program's execution, our tool relies on the results of the previously injected instrumentation functions. The tool provides API functions that are linked as dynamic libraries. These API functions receive the arguments' values of the original kernels that require a migration of their context. Our tool analyzes the allocated memory regions and finds the relation between these regions and the kernels' arguments. It defines the set of memory regions required for a kernel execution and provides these regions for serialization.

Memory analysis

Memory analysis is driven by the instrumentation functions inserted at compilation time. Our tool generates a memory map for the allocated regions. Each element in this map represents an allocated memory region with its attributes. These attributes contain the starting memory location, the allocation size, the data type size, and pointer locations in this memory region. These attributes are the metadata required for finding a kernel context and for serializing this kernel context.

The memory map is maintained by the instrumentation functions. For each memory allocation, a new element is inserted in the map. These elements are removed from the map if a tracked memory region is deallocated.

Relations between memory regions

Typically, the relations between memory regions are established with pointers that point from one region to another one. These connections represent object structures (or data structures) in memory.

If a kernel is automatically generated, typically a new function with arguments is generated. The tool uses the arguments' values of the original kernel to find the context required for the target kernel execution. These values are related to previously allocated memory regions and they can contain relations to other memory regions. Our tool finds these relations between the original kernel arguments' values and the allocated memory regions by searching the memory map (see Figure 3.3). The regions themselves can contain other pointers that must be followed. Therefore, the addresses stored in these pointers are also searched in the memory map to determine to what region they point. Recursively, our tool follows pointers from one region to the next one by searching the map. It is necessary to mark visited regions to prevent circles and to allow handling complex data structures like graphs. The set of determined memory regions that might be necessary for the kernel execution is prepared for migration.

Listing 3.2 shows the algorithm used to find the kernel contest. `argumentList` contains the arguments' value of the original kernel. Visited regions are marked by adding them to `visitedNodes`. `Pointers(object)` returns the set of pointers found in `object`. If a pointer is invalid or pointing to an object that has already been visited, there is not work to do, otherwise, the target object is marked and added to the `worklist`. All items in the `worklist` are processed until the `worklist` becomes empty. At the end, when `visitFromArgumentValues()` finishes, `visitedNodes` contains all the objects that might be necessary for the kernel execution. Objects that are not in `visitedNodes` are not required for the kernel execution.

Listing 3.2: The algorithm used to find the kernel contest.

```
visitFromArgumentValues():
    initialise(worklist)
    initialise(visitedNodes)
    for each argument in argumentList
        add(visitedNodes, arg)
        for each P in Pointer(arg)
            ref = *P  /* ref is the object pointed by P */
```

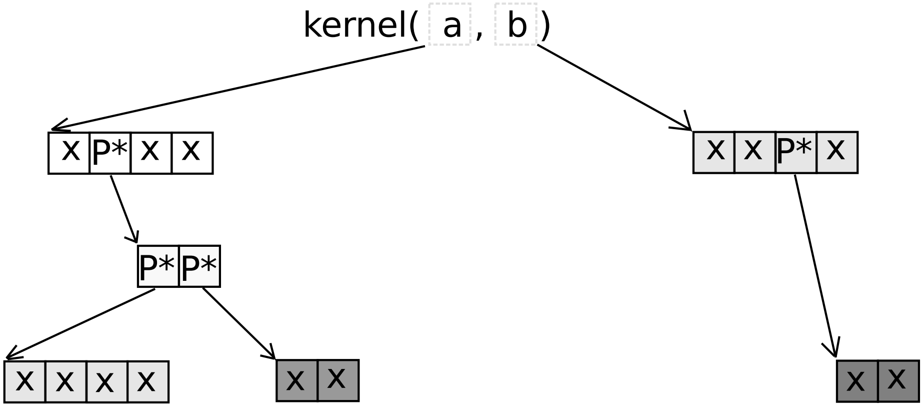


Figure 3.3: Values of a kernel arguments point to memory regions. Our tool finds the relation between these values and the allocated memory regions. It finds the set of memory regions required for the kernel execution.

```
            if ref != null && not hasItem(visitedNodes, ref)
               add(visitedNodes, ref)
               add(worklist, ref)
               visit()

initialise(worklist):
   worklist = empty

initialise(visitedNodes):
   visitedNodes = empty

visit():
   while not isEmpty(worklist)
      ref = remove(worklist)
      for each P in Pointer(ref)
         child = *P
         if child != null && not hasItem(visitedNodes, child)
            add(visitedNodes, child)
            add(worklist, child)
```

Automatic serialization

Once the set of memory regions required for kernel execution is known, this set is automatically serialized. Firstly, an array is created and all memory regions are copied into it. This array represents the serialized context. The size of this array equals the total size of all the memory regions in the set. Additionally, it is necessary to update the pointers inside the regions to preserve the connections between the copied memory regions. The pointers are updated with respect to their relative address in the array. If the expected starting address, where this array will be copied to the target resource, is known, these pointers are directly updated to the target address space. A mapping between the addresses of the memory regions on the host and their new addresses in the serialized context

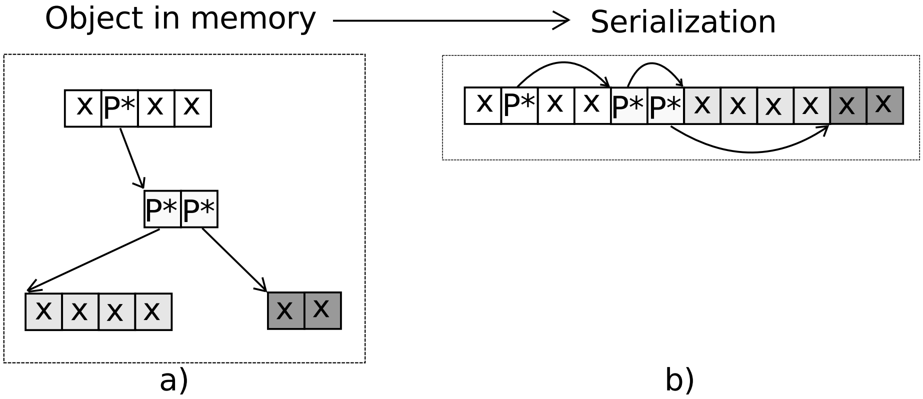


Figure 3.4: Memory regions required for kernel migration are serialized in an array. Pointers in the array are updated to preserve the connection between the serialized memory regions.

is saved. This mapping is used to calculate the updated values of pointers in the serialized context. The serialization of memory regions required for kernel execution is illustrated as an example in Figure 3.4.

Once the pointers are updated, the array now contains the complete serialized kernel context. The array is copied directly to the target memory so that the target kernel can be executed. After a successful execution of the target kernel, the array is copied back. Since other program elements can point to elements inside the kernel context, our tool deserializes the copied array and updates the original memory locations.

The advantage of this serialization approach is that it is completely generic. It handles the layouts of arbitrary data structures, as it relies on information in the memory map, which was constructed by the instrumentation functions.

As a corner case, the internal data representation of a kernel context may differ from one resource to another one (e.g., devices may have different pointer sizes). The current implementation of our tool collection does not support those corner cases.

3.4 Evaluation

In this section, we evaluate our tool collection by using different applications as benchmarks. First, we test if the automatically determined and serialized kernel contexts and the performed computations (which use these contexts) are correct. In our second test, we determine the overhead introduced to the application by our tool collection. Additionally, we analyze what conditions are necessary to compensate for these overheads. Finally, we verify the integration of our tool with other system elements. We integrate the modified applications in a simple resource-aware scheduler. This scheduler receives different resource-specific implementations of the same kernel. The scheduler relies on our tool for automatically generating the kernel context. At runtime, the scheduler decides which resource to choose according to the system requirements.

3.4.1 Evaluation Benchmarks

In this section, we describe the benchmarks used in the evaluation. These benchmarks were chosen because they cover a broad area of applications and have kernels that can take advantage of the chosen target resource in the evaluation.

Heat distribution (*HD*) The HD-application computes the temperature spread on a surface if a single point is heated. The algorithm computes the heat distribution on a discrete $n \times n$ field after T time steps. Every join of two edges in the grid represents a heat value that is set to the average heat of its four direct neighbors in each time step. We have two implementations for this algorithm. The first one uses a two-dimensional array to represent the heat distribution. We refer to it as *HD-n*. The second one uses a two-dimensional STL vector. We refer to it as *HD-stl*.

Markov chain (*MC*) A Markov chain is a system where the probability of its future states depends only on the current state, not on the past states. Stock/share price movements and the growth or the decline in a firm's market share [JI+12] are examples of Markov chains.

The benchmark performs repeated loops. In each loop a matrix A is multiplied by itself to produce A^2. Afterwards the L_1-norm of the difference between A and A^2 is computed. Matrix A is set in the next loop to A^2. The loop exits when the L_1-norm goes below a defined value or when the number of allowed iteration is reached.

In this application, a kernel is embedded in another kernel. While the inner kernel (the matrix multiplication) gains from the execution on a GPU, the outer one (the L_1-norm computation) performs best on the CPU. Thus, the kernel context is frequently moved between the CPU's and the GPU's memory.

Correlation Matrix (*CM*) Correlation is a statistical operation that finds the degree of the relation between two or more random variables. This algorithm has widespread applications, e.g., in finance, it is used to analyze stock and bond returns. In signal processing, it is used to find the header of data packets.

The benchmark computes the correlation between the columns of a given matrix. It outputs a matrix with elements between 1 and -1. Uncorrelated columns have 0 values. The correlation of a column with itself equals 1.

Dummy memory allocator (*DM*) The *dummy memory allocator* is an artificial benchmark instead of a real application. This program allocates a two-dimensional array in the heap memory and deletes it again. The program is executed multiple time to show the overheads introduced by the instrumentation functions.

3.4.2 Evaluation Environment

In our evaluation, we use two different systems equipped with different NVidia GPUs as target resources. The first system is a laptop equipped with an Intel i7

Application	Max result difference
HD-n	0
HD-stl	0
MC	$0.0126e-9$
CM	$59.6046e-9$

Table 3.1: The maximum difference between the results of the CPU kernels and their corresponding GPU kernels for the set of applications that we have tested. The small variation in the results of the *MC* and *CM* application results from the different order in the floating point operation caused by the parallelization.

(2860QM with 2.5 GHz) and a NVidia graphics adapter (NVidia Fermi Quadro 2000M, 2 GB DDR3). The second system is a server with a symmetric multiprocessing (SMP) computer node equipped with two Intel Xeon processors (E5620 with four cores 2.4 GHz each) and four NVidia Tesla accelerators (NVidia Fermi C2070, 6 GB DDR5).

3.4.3 Tool Evaluation

Correctness test

The following tests show that the automatically determined and serialized kernel contexts are correct and that the performed computations that use these contexts are correct.

In this test, we use the previously described benchmarks: *HD-n*, *HD-stl*, *MC*, and *CM*. We use the pluto compiler [BHRS08] with CUDA support as a code generator. We manually mark kernels in the original CPU codes of these benchmarks for the code generator tool. Then, we use this tool to automatically produce GPU kernels from the original CPU kernels.

First, we perform all computations with the original kernels. Afterwards, we perform the same computation with the tool generated GPU kernels. We use our tool collection to automatically identify, gather, and serialize the contexts of the original CPU kernel. The contexts are migrated from the CPU memory to the GPU memory. After all data is copied, the GPU kernel is executed. Once it terminates the context is copied back from the GPU's memory and automatically deserialized (see Figure 3.5). Finally, we compare the computations' results.

Our tests show that for all benchmarks the computed results are obtained equally from the original CPU kernel and its corresponding GPU kernel, since we use the same algorithms. This indicates the correctness of our tool collection. In Table 3.1, we show the maximum difference between the results of the CPU and the GPU kernels for the set of applications that we have tested. The small variation in the results of the *MC* and *CM* application results from the different order in the floating point operation caused by the parallelization.

Tool overhead test

The following test is done to determine the overhead introduced to the application by our tool collection. Additionally, we analyze what conditions must be met to compensate for these overheads.

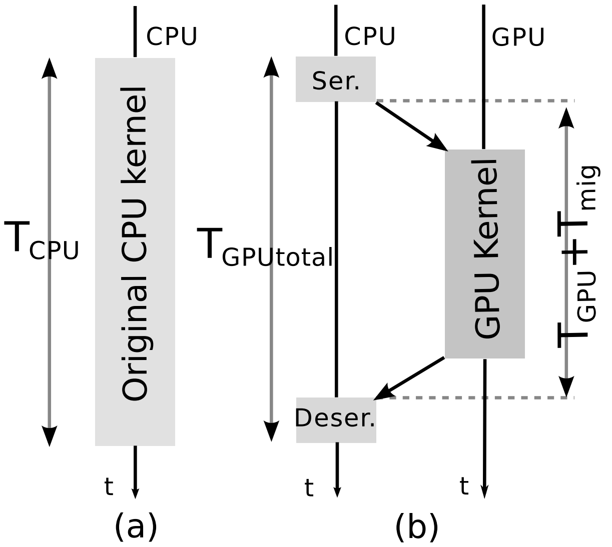


Figure 3.5: Subfigure (a) shows the execution of the original CPU kernel. Subfigure (b) shows the execution of the GPU kernel.

We partition this test into two subgroups: the serialization/deserialization overhead test and the instrumentation overhead.

Serialization and deserialization overhead test In this test, we use the same programs as before, which use the original kernels for computation and the tool-generated GPU kernels. We measure the following time intervals: (see Figure 3.5)

- T_{CPU}: time spent for executing the original CPU kernel without any instrumentation. No instrumentation functions inserted in the original kernel.
- T_{GPU}: time spent for executing the GPU kernel.
- T_{tool}: time spent for serialization and deserialization (the additional, accumulated times spent for identifying and serializing a kernel context, and the times spent to deserialize the context and copy it back to the original memory locations).
- T_{mig}: time spent for migrating the tool-generated context to and from the GPU memory.
- $T_{GPUtotal}$: sum of T_{GPU}, T_{mig} and T_{tool}.

We repeat all our measurements at least ten times and pick the median, maximum, and minimum values. The measurements are performed for all applications on both evaluation environments. For all applications, we provide three graphs that describe their execution properties.

The first graph shows the CPU execution time T_{CPU} and the total GPU execution time $T_{GPUtotal}$. Each graph's x-axis shows the applications' problem sizes and each graph's y-axis shows the required times in seconds. In each graph, both axes are drawn in logarithmic scale.

The second graph for each application displays the extra time introduced by serialization and deserialization (denoted T_{tool}). Each graph's x-axis shows the applications' problem sizes in logarithmic scale and each graph's y-axis shows the required times in seconds in linear scale.

The third graph displays the $speedup = {}^{T_{CPU}}/_{T_{GPUtotal}}$ for each application. Each graph's x-axis shows the applications' problem sizes in logarithmic scale and each graph's y-axis shows the achieved *speedups* in linear scale.

In all graphs, we show the median values of our measurements. In selected cases, we show the maximum and minimum (e.g., if the variance in the measurements is high).

Heat distribution (HD) The heat distribution application computes the temperature changes in a two-dimensional grid. The grid's side length is given as N, which defines the problem size. We provide two implementations for this application: *HD-n* and *HD-stl* While the *HD-n* application uses standard C data structures, the *HD-stl* application uses also C++ data structures provided by the STL.

For the *HD-n* implementation on both systems, we notice for small N that T_{CPU} is smaller than $T_{GPUtotal}$. As N increases, the execution time on the CPU T_{CPU} increases faster than the execution time on the GPU $T_{GPUtotal}$ (see Figure 3.6). These results show that for medium sized and large values of N the speedup caused by the GPU compensate for the overhead T_{tool} introduced by our tool collection. This overhead T_{tool} can be calculated and shown by the difference between $T_{GPUtotal}$ and $T_{GPU} + T_{mig}$. The values of T_{tool} increase by increasing N (see Figures 3.7). However, there is some noise in the measurements on the Intel Xeon system (see Figures 3.6 and 3.7). This noise is caused by the first executions of the GPU kernel. In the first runs, the serialization takes longer than the later ones. That is due to the fact that the first access to the GPU driver on the Intel Xeon system has high variance and takes longer than later acesses. The minimum and maximum times shown in Figure 3.7 display the values' wide variation. Thereby, the maximum values vary from one measuring point to the other, while the minimum values form a continuous curve (similar to the i7 measurements). However, if the first executions are not included in the time measurements, these variations nearly disappear.

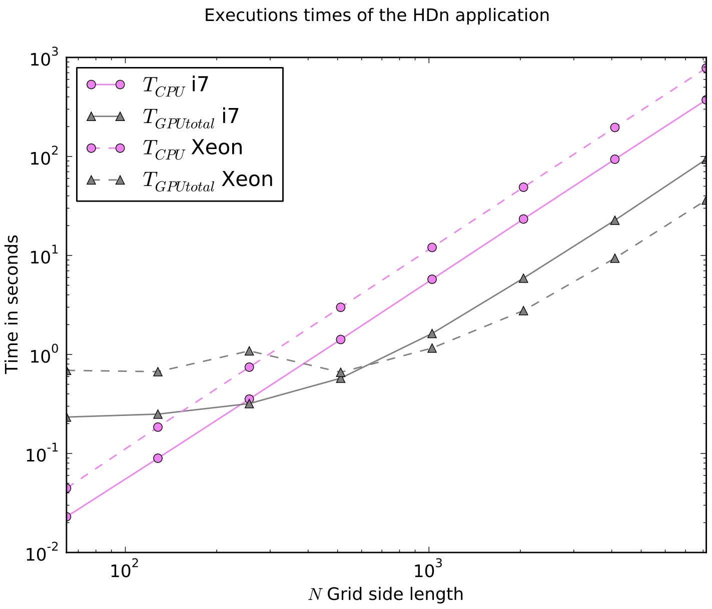


Figure 3.6: The execution time results of the *HD-n* application using an Intel i7 and an Intel Xeon.

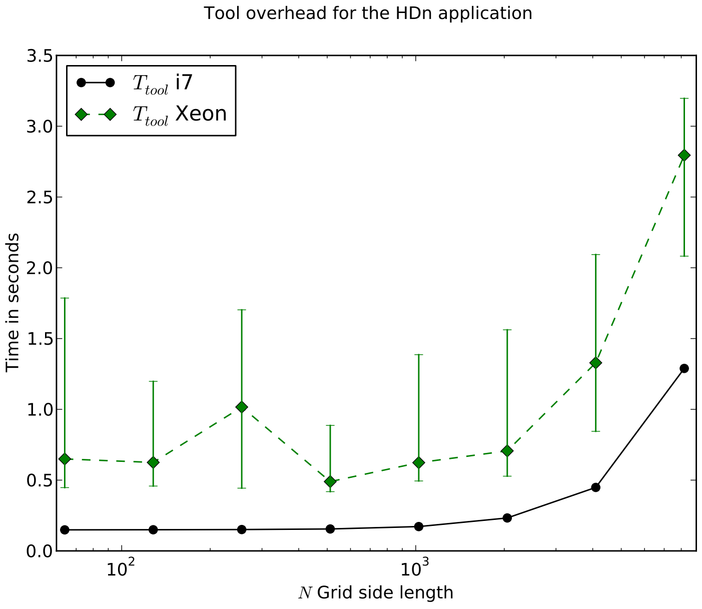


Figure 3.7: The serialization and deserialization overhead of the *HD-n* application using an Intel i7 and an Intel Xeon introduced by our tool collection.

The *speedup* is the ratio between T_{CPU} and $T_{GPUtotal}$. In the HD applications, the *speedup* becomes greater than one once T_{CPU} becomes greater than $T_{GPUtotal}$ (see Figure 3.8). The *speedup* on the Intel Xeon system is higher than on the Intel i7 system. This can be explained by the different hardware configurations. A single core of the Intel i7 is faster than a single core of the Intel Xeon, while the GPU (Tesla accelerator) on the Intel Xeon system in faster than the GPU (graphics adapter) on the Intel i7 system. This can also be seen in Figure 3.6.

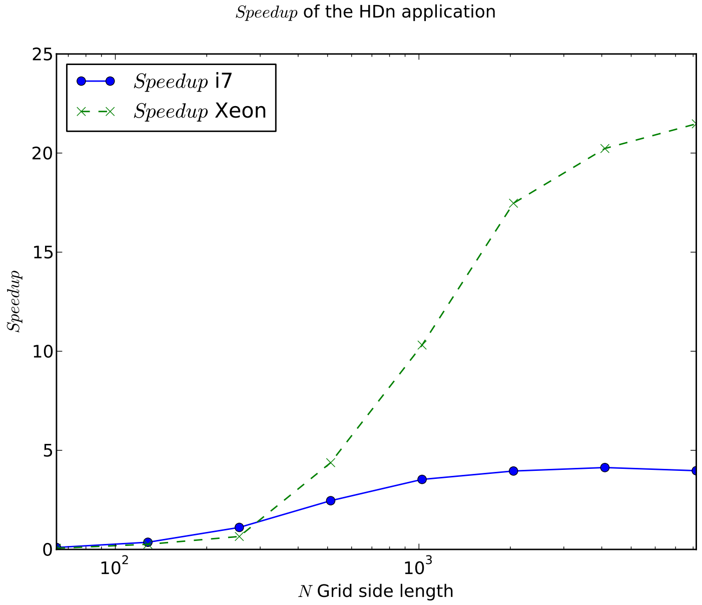


Figure 3.8: The *speedup* of the *HD-n* application using an Intel i7 and an Intel Xeon.

In general, the used CPU kernels are neither vectorized nor parallelized codes which run on a single core, while the GPU kernels are automatically generated and exploit the GPU advantages. The GPU has parallel cores that parallelize the execution of the GPU kernel. This parallelization reasons the speedup on the GPU. However, there are a limited number of GPU cores. The Intel Xeon system's GPU (Tesla accelerator) has more cores than the Intel i7 system's GPU (graphics adapter). This explains why the speedup saturates earlier on the Intel i7 system than on the Intel Xeon system (see Figure 3.8).

However, we notice that on the largest value of N the *speedup* on the Intel i7 system decreases by 3% from its previous value. This can be explained by the increase of the time spent by the tool overhead T_{tool}. However, this additional costs can still be balanced by the kernel acceleration.

In general, copy operations for the serialization and deserialization of arrays requires less effort than the same operations for STL-containers. However, STL-vectors can still be easily handled like arrays. Copying the contents of vectors and arrays requires almost the same effort. The additional computation times

T_{tool}, which are introduced by these additional operations, can be seen in Figure 3.9 for the *HD-n* and the *HD-stl* applications on the Intel i7 system. In both cases, T_{tool} increases when N is increased. Since the *HD-stl* application uses vectors, the time spent for copying data is almost the same as in the *HD-n* application.

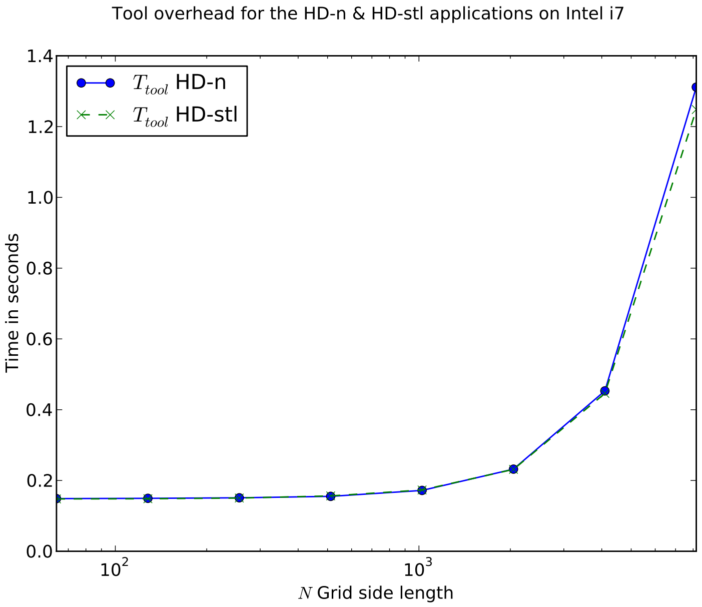


Figure 3.9: The serialization and unserialization overhead differences of the *HD-n* and *HD-stl* applications using an Intel i7.

Markov chain (MC) The Markov chain application performs a continuous multiplication of a square matrix ($N \times N$). N is the matrix side length, which defines the problem size.

We notice that in both evaluation systems T_{CPU} is greater than $T_{GPUtotal}$ for all values of N (see Figure 3.10). Thus, the speedup gained by utilizing the GPU compensates the overhead T_{tool} introduced by our tool collection. The values of T_{tool} increase by increasing N (see Figure 3.11). However, we do not experience any noise like in the *HD*.

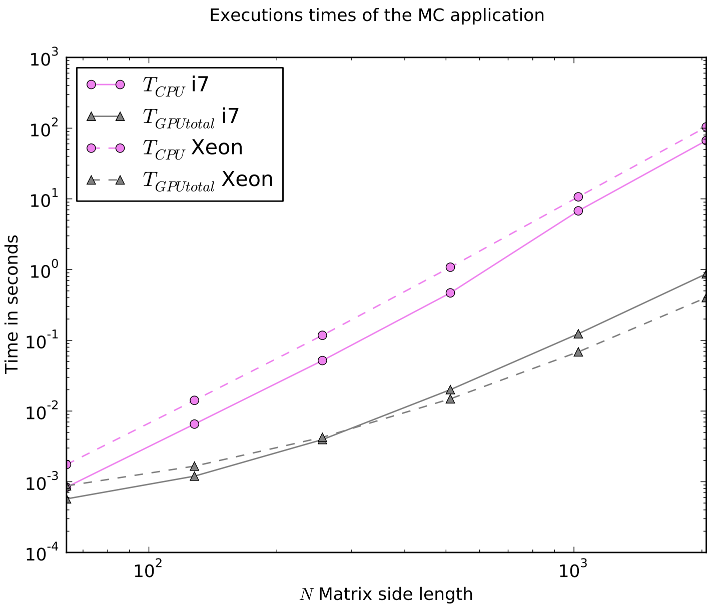


Figure 3.10: The execution time results of the *MC* application using an Intel i7 and an Intel Xeon.

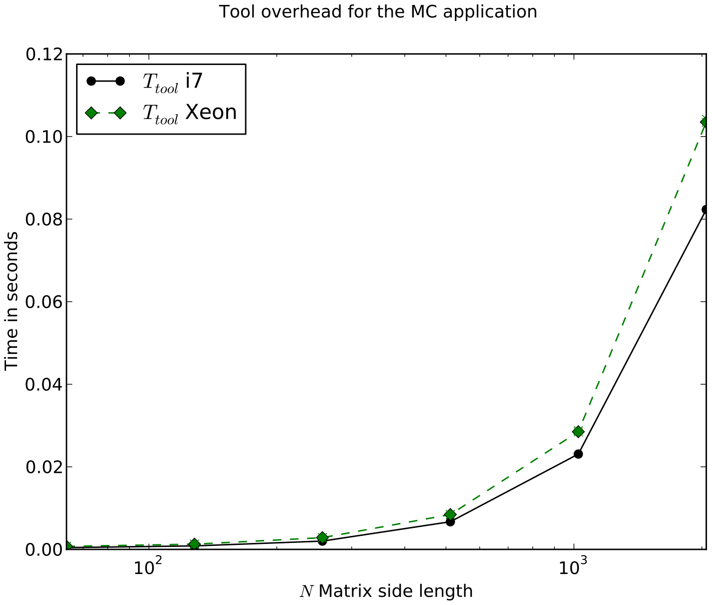


Figure 3.11: T_{tool}, the serialization and deserialization overhead, of the *MC* application using an Intel i7 and an Intel Xeon.

For the *speedup*, we have a similar behavior like in the *HD*, but without any saturation (see Figure 3.12).

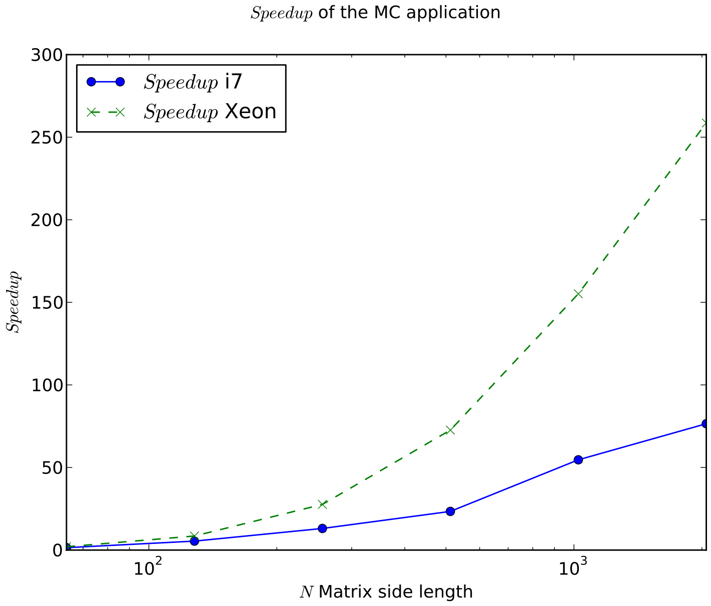


Figure 3.12: The *speedup* of the *MC* application using an Intel i7 and an Intel Xeon.

Correlation Matrix (CM) The Correlation Matrix application computes the correlation between the columns (N) of a given matrix. The number of columns in this matrix is denoted as N, which defines the problem size.

For both systems, we notice that T_{CPU} is less than $T_{GPUtotal}$ for small N values. As N increases, T_{CPU} becomes greater than $T_{GPUtotal}$ (see Figure 3.13) and the tool overhead T_{tool} is compensated by the GPU speedup. This behavior is similar to the case of *HD*.

The values of T_{tool} are similar to the values in *HD*. Again, we experience some noise (see Figure 3.14). Also the *speedup* of *CM* behaves similar to the *speedup* of the *HD* applications (see Figure 3.15). However, there is no saturation and only for large N values is the *speedup* on the Intel Xeon larger than i7.

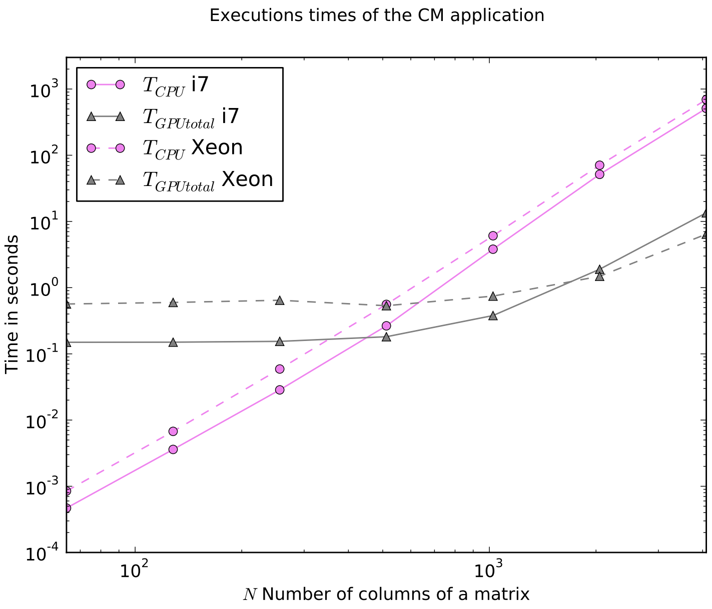


Figure 3.13: The execution time results of the *CM* application using an Intel i7 and an Intel Xeon.

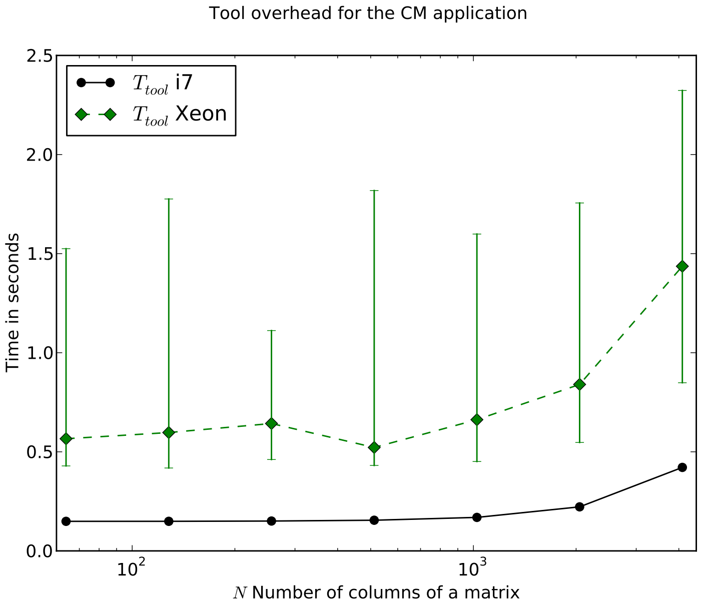


Figure 3.14: T_{tool}, serialization and deserialization overhead, of the *CM* application using an Intel i7 and an Intel Xeon.

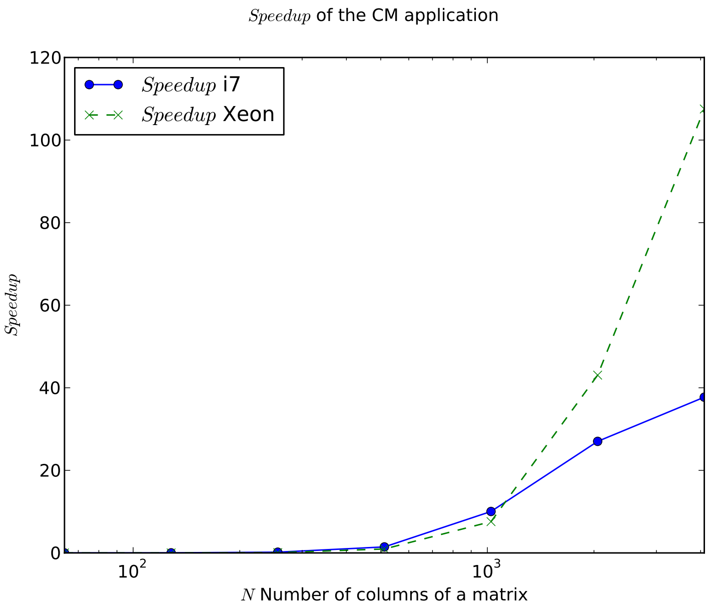


Figure 3.15: The *speedup* of the *MC* application using an Intel i7 and an Intel Xeon.

All applications on all systems have T_{CPU} greater than $T_{GPUtotal}$ for sufficiently large values of N. The Markov chain application even shows T_{CPU} greater than $T_{GPUtotal}$ for all values of N.

The overhead T_{tool} introduced by our tool collection can be compensated by the speedup introduced by the GPU. In this test, we did not show the quality of the code generators. Instead we have shown how the speedup introduced by the automatically generated kernels can compensate the overhead of our tool collection.

Instrumentation overhead test Overhead is introduced to the application by the instrumentation functions that have been inserted to the original program by our LLVM compiler-pass at compilation time. To determine these additional costs, we perform an artificial test (the *DM* benchmark) where only the management of the actors is measured. We compile two version of this benchmark with the same optimizations: one with instrumentation and one without instrumentation.

To determine the overhead, we measure the time to allocate and delete two-dimensional arrays ($N \times N$) in the heap for both applications. We repeat all our measurements hundred times and pick the median value. The difference in the execution time determines the additional time required for instrumentation.

In these tests, N is the side length of a two-dimensional array, which defines the allocation size. The results in the difference of the execution time is shown in Figure 3.16 for an Intel i7 and an Intel Xeon system. The x-axis shows the arrays' side length N in logarithmic scale and the y-axis shows the time difference in seconds in linear scale.

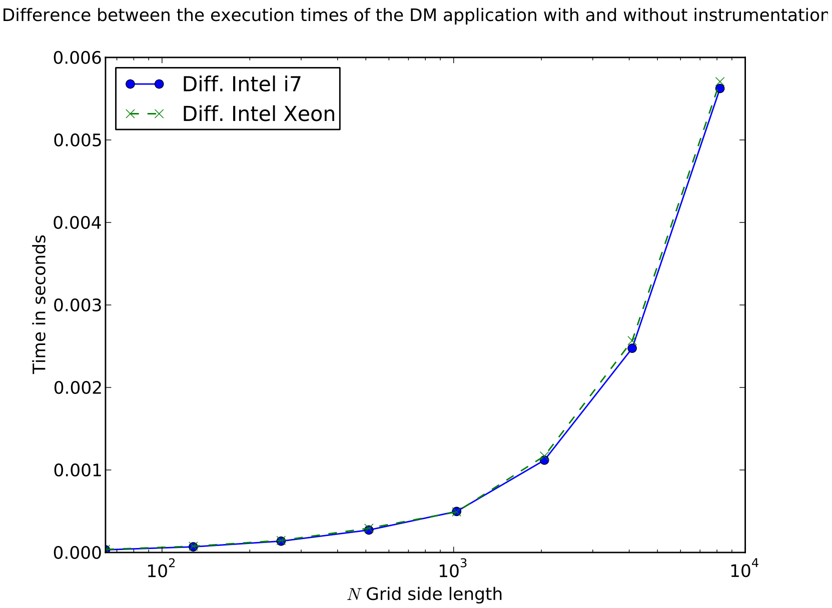


Figure 3.16: The difference in the execution time of the *DM* application with and without instrumentation using an Intel i7 and an Intel Xeon system.

The results show that the instrumentation overhead increases by increasing N. This overhead is in the order of milliseconds and, thus, can be neglected with respect to total program execution time.

Tool integration

We verify the integration of our tool with other system elements. We integrate the modified applications in a simple resource-aware scheduler (*VarySched*) [SDG+16a] [SDG+16b]. We have developed *VarySched* as a configurable task scheduler framework tailored to efficiently utilize all available computing resources in a system. *VarySched* allows a more fine-grained task-to-resource placement which is even further enhanced by allowing the tasks to migrate to another resource during their runtime. In addition, *VarySched* can manage multiple scheduling strategies, optimizing (for instance, throughput or energy efficiency) and switch between them at any time.

Figure 3.17 shows a block diagram of *VarySched*. It receives a patch of kernels with their context information. Each kernel can have multiple implementations: a single-core kernel, an OpenMP multi-core kernel, a GPU kernel. The scheduler binds the received kernels to all resources available on its node and external nodes. The scheduler relies on our tool in automatically generating the kernel context so that the scheduler can support running GPU kernels and dynamic kernel migration between resources at runtime.

The scheduler (*VarySched*) provides a shared memory region where applications can register their kernels for different resources at runtime. The functions to register a kernel and the necessary context-copy functions are provided by the scheduler's library. In combination with the resource specific kernel, an affinity of the kernel to the resource must be submitted to the scheduler. After the different kernels of an application have been registered, the scheduler informs the application when its kernel is delayed (since all resources are in use), when to start its kernel, and when its kernel must be moved to another resource. After a kernel has performed its calculation for a while, it can pause itself and ask the scheduler if it has to leave its current resource, if a better resource is available or if it has to leave the resource (e.g., since another application is scheduled on the resource). In general, since a kernel can be migrated from one resource to another at runtime, the scheduler must migrate the applications' contexts. Due to this, applications can start their computations on one (not that ideal) resource and proceed later on the preferred one, instead of waiting until its preferred resource is free. Thus, the scheduler relies on our tool in automatically generating the kernel context. At runtime, the scheduler decides which resource to choose according to the system requirements. For the scheduler, we generated three different kernels: a single-core kernel, an OpenMP multi-core kernel, and a GPU kernel (see Figure 3.17). We started one hundred instances of our benchmark applications in parallel with the resource-aware scheduler and compared the result with a sequential start of the applications which uses only the GPU for its computations. We have also tried to start one hundred instances of the application in parallel without the scheduler but this test crashed our computer.

Figure 3.18 shows that the total execution time of the programs can be reduced if the scheduler is used. However, to achieve a reduction of the execution time, the resource affinities of the applications (and in this way the distribution among the devices) must be well chosen. In our case, the metric to determine the affinity is the relation between the execution time on the GPU and the execution time on a single core of the CPU.

3.5 Conclusion

In this work, we presented a tool collection *ConSerner* that automatically identifies, gathers, and serializes the context of an original CPU kernel and migrates it to a target resource's memory where a target specific kernel is executed with this data. The overhead introduced by our tool can be compensated by exploiting the advantages of the target resource over the ordinary CPU. We have also shown that the usage of a resource-aware scheduler can help to use all resources efficiently.

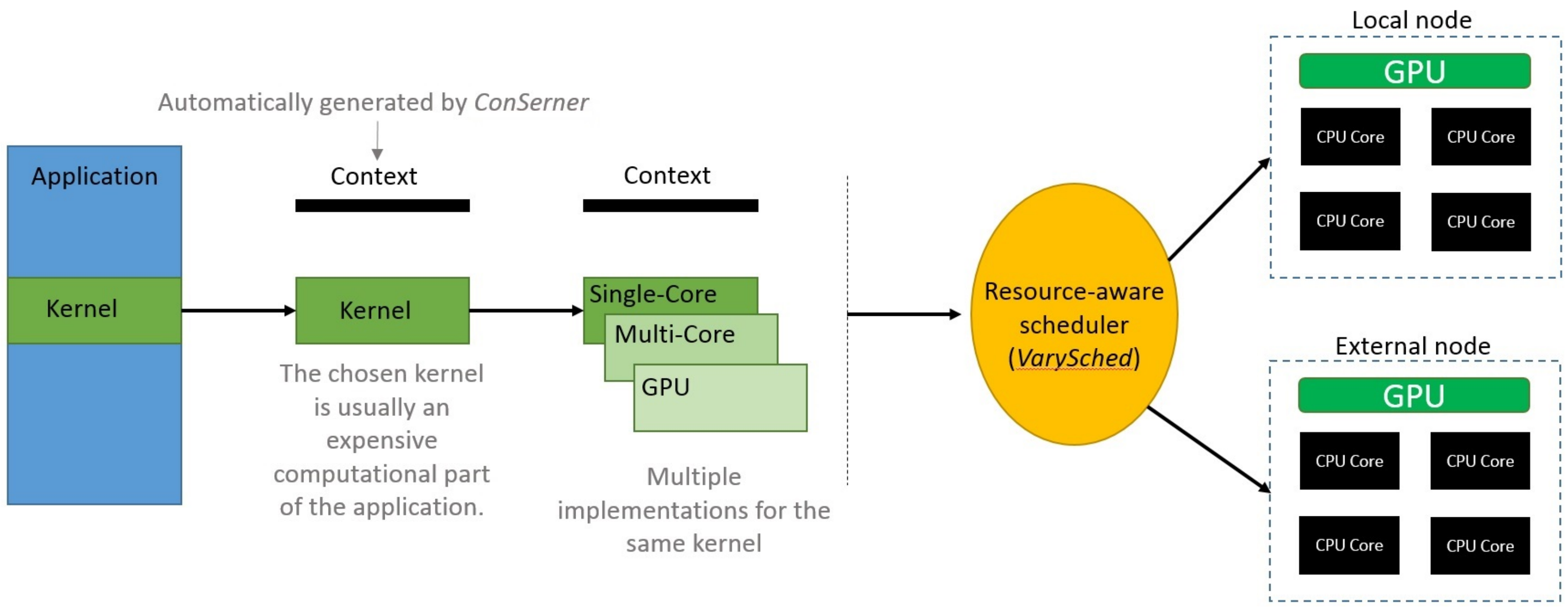

Automatically generated by ConSerner
Application
Kernel
Context
Kernel
The chosen kernel is usually an expensive computational part of the application.
Context
Single-Core
Multi-Core
GPU
Multiple implementations for the same kernel
Resource-aware scheduler (VarySched)
Local node
GPU
CPU Core
CPU Core
CPU Core
CPU Core
External node
GPU
CPU Core
CPU Core
CPU Core
CPU Core

Figure 3.17: A block diagram of a simple resource-aware scheduler (*VarySched*).

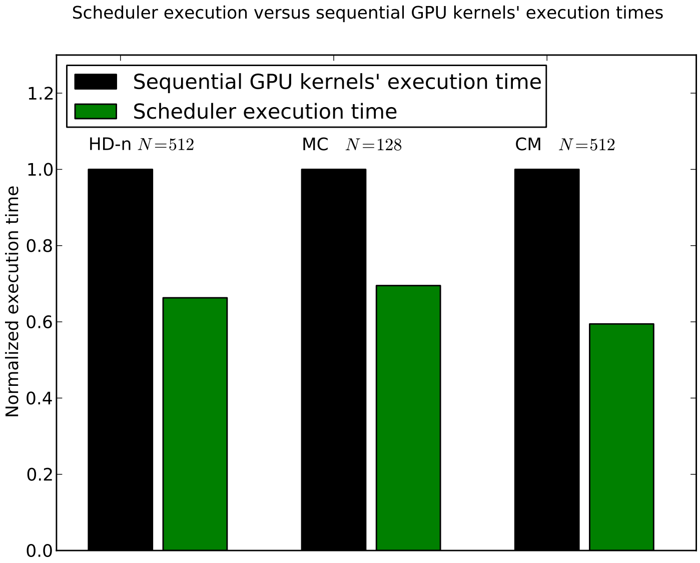


Figure 3.18: The execution time of one hundred instances of the *HD-n*, *MC* and *CM* applications. All values are normalized with respect to the total execution time without the scheduler.

Chapter 4

Zeroing Memory Deallocator to Reduce Checkpoint Size and Accelerate Migration in a Virtualized HPC Environment

The migration of jobs across the cluster is necessary to support dynamic load balancing. A migration mechanism can also help to improve the systems' resiliency, as in the case of imminent failures an evacuation of affected nodes can be performed by a migration of the respective processes [WMES12,NMES07]. In previous studies, we investigated different migration techniques and found full virtualization based on KVM [KKL$^+$07] to offer high flexibility, while providing performance results comparable to a native execution [PGL$^+$14]. A drawback of this approach is high migration times caused by the transfer of partly unused and therefore unnecessary memory regions. The transfer of unnecessary memory regions 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, The transfer of unnecessary memory regions 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 chapter, we propose an approach to decrease the storage size of HPC applications' checkpoints and to accelerate application migration in HPC by reducing the transmitted data volume. We showcase our approach by taking the example of VM migration/checkpointing. The migration time of VMs is mainly determined by the network bandwidth [HGLP07] 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 [STHE11] and zero-block detection [DS11]. 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. We demonstrate that our 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. We also show that our approach reduces the checkpoint size of our tested applications by up to 9 % without compression and by up to 94 % with compression.

The remainder of this chapter is structured as follows: After discussing related work, we explain our approach in Section 4.2. In Section 4.3, we present a comprehensive evaluation of our approach before generalizing our approach in Section 4.4 and concluding this chapter in Section 4.5.

4.1 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 [NMES07]. They monitor the health of computing nodes in order to detect deteriorating behavior and 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 [MNB$^+$15,HGSZ10]. Randles et al. present a comparison of distributed load balancing strategies [RLT10].

Different migration techniques exist in HPC, such as, *process-level migration*, *virtual machine migration* and *container-based migration* [PGL$^+$14]. Process-level migration is based on the checkpoint/restart (c/r) mechanism, which allows an application to save a snapshot of its current state, so that it can be restarted from that point 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 [Due03]) or in user space (see DMTCP [AAC09]). The advantage of system-level checkpoint/restart is that it is transparent to the application and that the checkpoint can be taken at arbitrary points. However, these tools produce relatively large checkpoints because they include data that is not required for restarting the computation.

Application-level c/r was introduced to get checkpoints of smaller size, but it is also more complex and involves the application programmer. The programmer must know the data structures to be included into the checkpoint and has to add this information to the code. In order to ease the process, the programmer is assisted 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 [PBKL95]. 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 [BMP$^+$04] and MPI programs [SBF$^+$04]. An approach to reduce the checkpoint size of application-level Checkpoint / Restart (CR), which is orthogonal to the approach presented in this chapter, is to deduplicate different generations of checkpoints [KGS$^+$16].

The virtualization overhead is often seen as the main reason why virtual machine migration is rarely used in HPC. Youseff et al. show that this overhead can be neglected and that the performance of virtual machines is relatively close to native execution [YWGK06]. Pickartz et al. demonstrated that virtual machine migration can also be beneficial in HPC because of its flexibly and even improved performance [PGL$^+$14].

A lot of effort is spent on accelerating these virtual machine migrations through focusing on increasing the bandwidth between source and destination nodes, and 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 RDMA (Remote Direct Memory Access) over InfiniBand [HGLP07]. In this way, they are able to increase the available bandwidth for migration and reduce the migration overhead by 80% with respect to TCP/IP. Satyanarayanan et al. propose a suspend/resume approach for virtual machines, in which a suspended virtual machine saves its volatile state to a file [SG$^+$07a, KS02]. This file is copied to a remote node where the virtual machine can be resumed.

Live migration is a technique for moving virtual machines between computing nodes with almost zero downtime. There are different techniques for live migration, such as *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 [HNIS11]. 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 [CFH$^+$05]. As this memory might get updated after being copied from the host node, updated memory pages are iteratively copied to the source node before finally the process state can be copied to the target node. Precopy works better with read-intensive applications. With precopy, write-intensive applications accessing large amounts of memory can make migration impossible [HNIS11]. Precopy is also more resistant to faults because the source node still holds an updated copy of the virtual machine. On the other hand, postcopy typically experiences the shortest downtime.

4.2 Methodology

This section describes our proposed solution for an accelerated migration of VMs in detail. First, we give an overview of the migration mechanism inside QEMU / KVM. Afterwards, we introduce our preload library implementing the zeroing of freed memory. Finally, we provide a brief description of selected HPC application benchmarks that we use for an evaluation of the presented approach.

4.2.1 Virtual Machine Migration in QEMU / KVM

KVM is an open source Linux-based virtualization solution [KKL+07]. It provides full virtualization on x86 hardware utilizing the VT-x or AMD-V hardware extensions [UNR+05, Vir05] 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 [Bel05]. 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, introduced by Clark et al. [CFH+05], 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 an retransmitted, 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 guests 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 or not. 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 implemented 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.

4.2.2 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 (See Figure 4.1) 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, 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 (See Figure 4.2). The flags encode whether the current 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

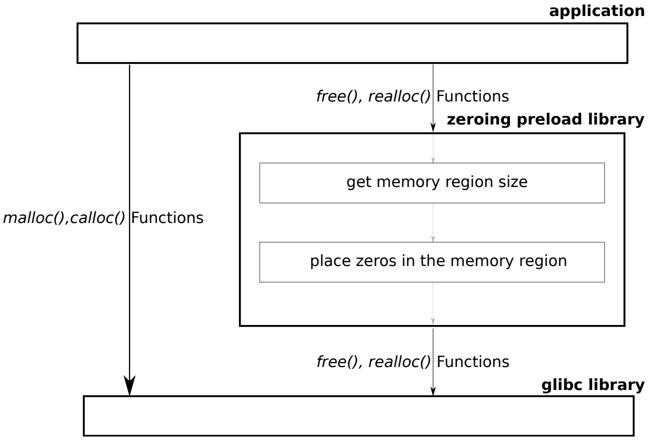

application
free(), realloc() Functions
zeroing preload library
get memory region size
place zeros in the memory region
malloc(),calloc() Functions
free(), realloc() Functions
glibc library

Figure 4.1: The preload library intercepts deallocation operations issued by an application.

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 reasons 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 [NB10, Fer07].

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. Furthermore, this overhead has to be small, so that the zeroing of regions, which get reallocated and rewritten by the application, does not lead to increased runtimes. As shown in Section 4.3, 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.

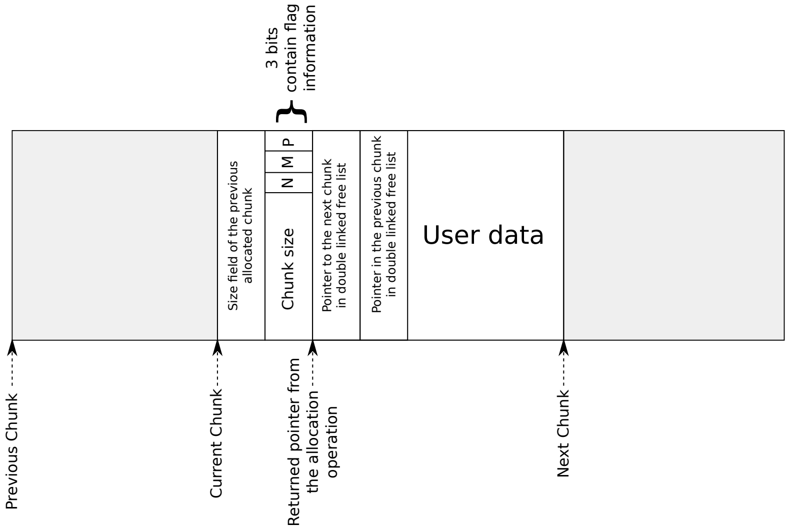

3 bits
contain flag
information
Size field of the previous allocated chunk
N
M
P
Chunk size
Pointer to the next chunk in double linked free list
Pointer in the previous chunk in double linked free list
User data
Previous Chunk
Current Chunk
Returned pointer from the allocation operation
Next Chunk

Figure 4.2: The structure of an allocated chunk managed by glibc [Fer07]. Only deallocated chunks are connected to a double linked free list.

4.2.3 HPC Application Benchmarks

The following is a brief description of the HPC applications which we have tested with our approach:

mpiblast [DCF03] is a parallel implementation of NCBI BLAST and computes alignments of DNA sequences. It scales to hundreds of processors and improves NCBI BLAST's performance by several orders of magnitude. mpiblast achieves this by using database fragmentation, query segmentation, intelligent scheduling, and parallel I/O. In our experiments, we compared 10,000 nucleotide sequences with microbial sequences from the MBGD database [Mbg].

NAMD [PBW$^+$05] is a parallel and highly scalable code designed for the simulation of large biomolecular systems. It is written using the Charm++ [KK93] programming framework. It uses a combination of spatial and force decomposition to benefit from the available processors.

gromacs [HKVDSL08] is a widely used software to perform molecular dynamics (MD) simulations of molecules, like proteins and lipids, in which complex bonded interactions are involved. We ran a program that calculated the absolute solvation free energy of ethanol.

The system consisted of an ethanol molecule in a box of 895 water molecules. The input data was spread over different files and contained information about the system, like the number of atoms in the system, the type of each atom, the coordinates of each atom in nanometers, and the dimensions of the simulation box (2.99491, 2.99491, 2.99491) in nanometers. Additionally, there was a topology file which contained information about the topology of the system being simulated and a file which contained simulation parameters, like the number of steps of the simulation and the integration interval. At the beginning of the simulation, the simulation box was divided among the available processors via neutral-territory domain decomposition using MPI plus OpenMP multi-threading within a rank.

phylobayes [LLB09] is a Bayesian Monte Carlo Markov Chain sampler for phylogenetic reconstruction using protein alignments. In our experiments, we used the alignment of the PB2 gene of 401 influenza viruses. We used a stick-breaking Dirichlet process mixture with the following characteristics: the number of taxa is 401, the number of sites is 2277 and the number of states is 4.

LAMMPS [Pli95] is also a molecular dynamics program. By default it uses spatial decomposition with equal-size domains. It also implements a series of commands that can augment the distribution of the domains among processors based on different criteria. In addition to exploiting inter-node parallelism via the aforementioned decomposition, LAMMPS also exploits intra-node parallelism by using variety of packages (GPU, USER-CUDA, USER-OMP, USER-INTEL, KOKKOS).

We ran the ReaxFF benchmark, which is a simulation of PETN crystal, and is shipped with the software package. The system under study consisted of 58 atoms of four different types and the data file contained the coordinates of all the atoms.

4.3 Evaluation

In this section we evaluate the proposed approach in terms of runtime overhead and checkpoint size. We show that the reduction of the checkpoint size descreases the migration time.

We have used two NUMA nodes for the evaluation. Both nodes have 32 virtual cores on two sockets with 8 physical cores each. The nodes are equipped with Intel SandyBridge CPUs (E5-2650) clocked at 2 GHz and connected by a Gigabit Ethernet fabric. Both systems have the same software stack and run an unmodified CentOS 7.2 installation with a 3.10.0 Linux kernel. The virtualization framework is based on KVM and QEMU version 2.5.1.

4.3.1 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. For this reason, it is highly application-dependent: it depends on the number of times memory is freed and the size of the affected memory regions. To assess this overhead, we compared the execution time of each application with and without the preload library.

We took the median over ten runs to obtain stable values. PhyloBayes was excluded from the test because its execution time lasts longer than four days. Figure 4.3 shows the execution times of our test applications, which are normalized with respect to the median. mpiblast exhibits a large preload library overhead of 7.1 %. 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.

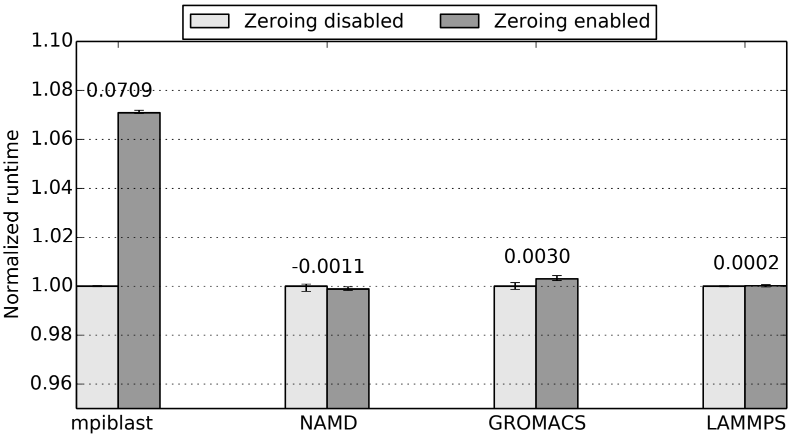


Figure 4.3: Impact of the preload library on the runtimes of selected HPC applications. The bars represent the normalized execution time, the median and the upper and lower quartiles. The numbers above the bars are the difference between the medians for enabled and disabled zeroing.

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

Source of the Overhead

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 deallocation operations issued by the application at runtime. We ran each test application inside of a VM with the modified preload library generating the trace. Figures 4.4 and 4.5 show the size of memory set to zero in bytes per minute, and the number of deallocation operations per minute issued by the application respectively. (Every value in the graphs represents the accumulated amount of the previous minute.)

mpiblast shows the highest amount of memory set to zero per minute (See Figure 4.5) and has the highest deallocation rate after LAMMPS (See Figure 4.4) which explains why mpiblast has the highest preload library overhead in Figure 4.3. This is probably the result of a frequent usage of communication buffers within the MPI layer. However, further investigations are necessary to validate this assumption. LAMMPS, in contrast, reveals the lowest amount of memory set to zero per minute. For LAMMPS, some points are missing in Figure 4.5 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

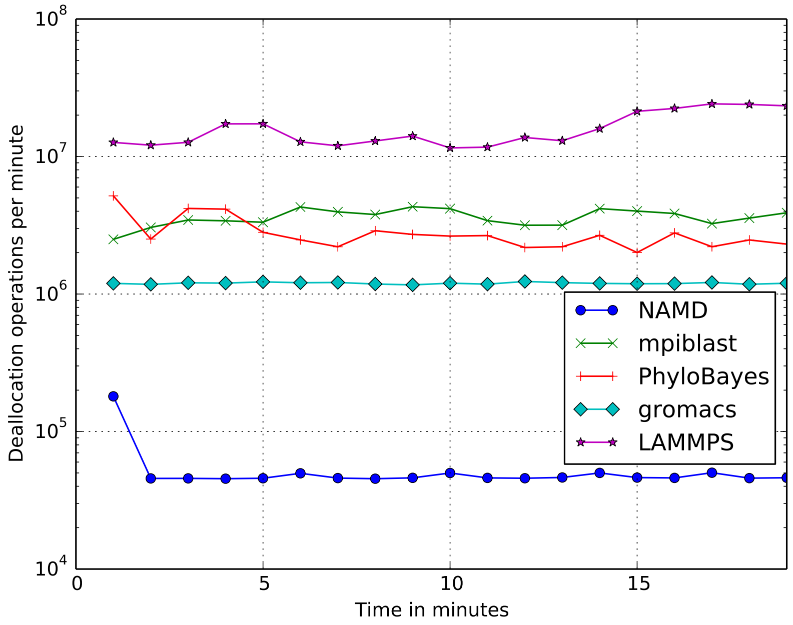


Figure 4.4: The number of deallocation operations intercepted by the preload library per minute.

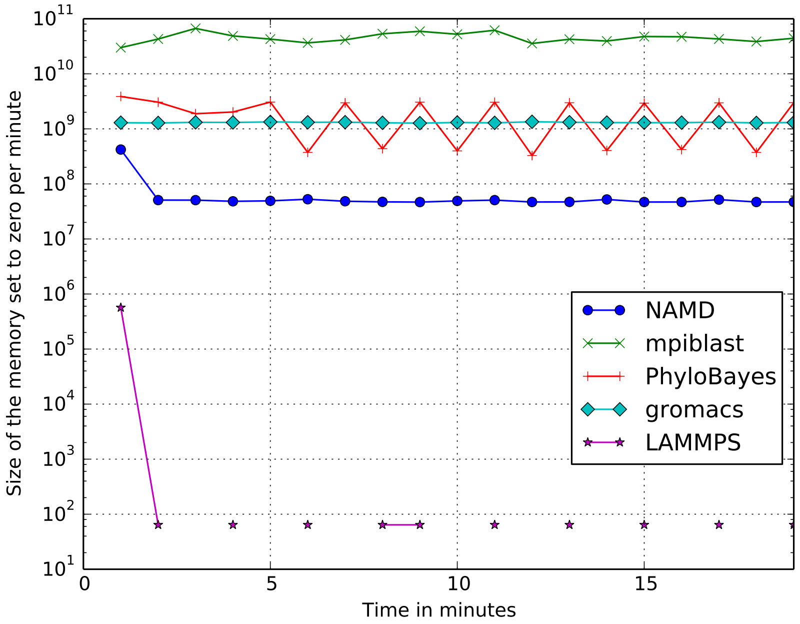


Figure 4.5: The amount of data in bytes zeroed by the preload library per minute.

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.

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.

4.3.2 VM Image Size

The migration time of a running application inside a VM is affected by the size of the VM image, which mainly depends on the size of the application's memory image. We studied the effect of our zeroing preload library on the size of the VM image and examined whether compression and zero-page detection algorithms benefit from it. Since KVM performs checkpointing as a memory core dump, a checkpoint 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 5 minutes intervals and compressed them logging the checkpoint size before and after compression.

The zero-page detection algorithm is applied by the hypervisor with every checkpoint. So we log the number of these detected zero pages. Note that the first complete zero pages are discarded by the zero-page detection algorithm, after that compression is applied to get rid of partial zero pages.

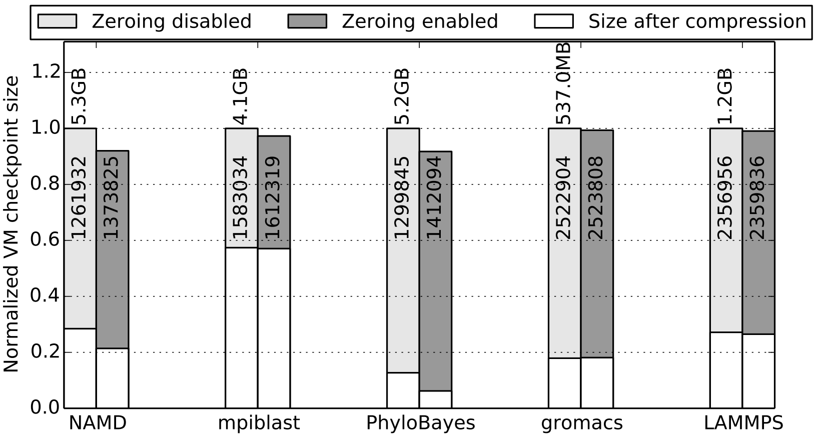


Figure 4.6: Impact of the preload library on the checkpoint size. The values are normalized to the checkpoint size with disabled preload library and without compression. The values above the bars are the absolute size with disabled preload library and without compression. The values inside the bars are the number of detected zero pages in the checkpoint image.

Observing the results in Figure 4.6, we see that, with compression disabled, all applications have a smaller checkpoint size when zeroing is enabled. Also all applications experience a larger number of zero pages when zeroing is enabled. With compression enabled, the checkpoints of all application, except gromacs, are smaller when zeroing is enabled.

From the results we can derive that, for all applications, the zero-page detection algorithm found additional zero blocks generated by the preload library. The compression algorithm is applied after the zero-page detection algorithm. For all applications, except gromacs, compression benefitted from the additional partial zero pages generated by the preload library.

Zeroing is able to decrease the checkpoint size by up to 9 % without compression and by up to 94 % with compression, with respect to the case with disabled preload library and without compression. The benefit of zeroing depends on the number of full/partial zero memory blocks detected within the application at the time of the checkpoint.

4.3.3 VM Migration Time

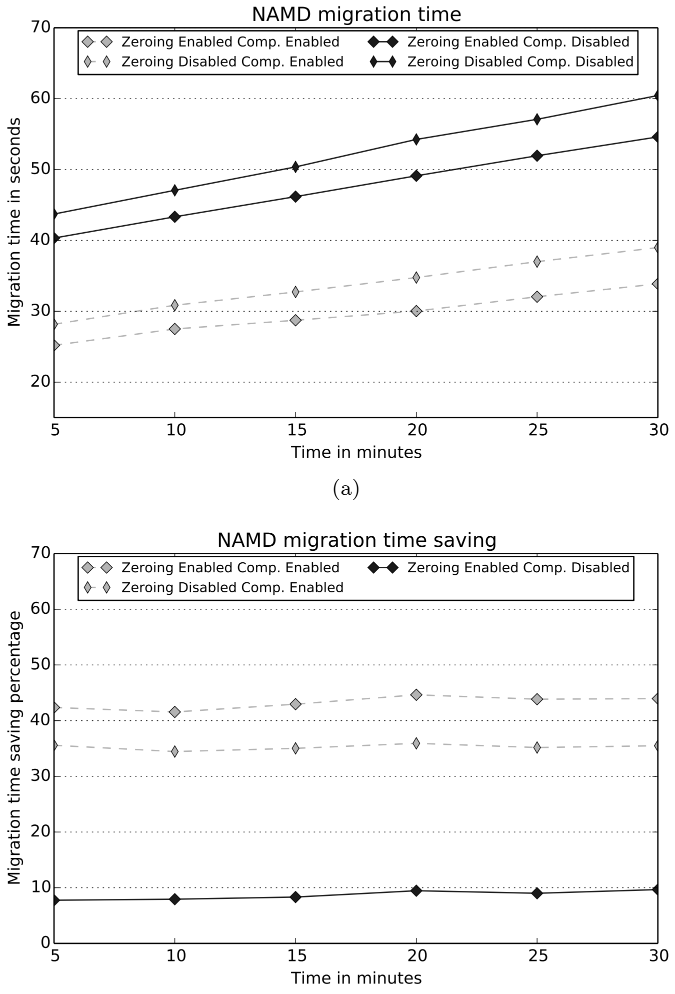
One of the goals of our zeroing approach is to accelerate application migration. For this reason we compare the migration times of HPC application benchmarks with and without the preload library enabled. For this experiment, we selected NAMD, PhyloBayes and mpiblast because, looking at the previous results, one would expect them to benefit the most from our approach.

We ran each of the three applications for more than half an hour and migrated them back and forth between the two cluster nodes at intervals of 5 min. Each application started in a VM with 10 GiB of guest physical memory. The VM's virtual CPUs were mapped to the host's topology [BPWM16]. To do 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 exaclty 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 averaged the results. 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 Figures 4.7, 4.8 and 4.9, zeroing accelerated the migration time for all applications regardless of whether compression was enabled or not. When zeroing was applied alone without compression, NAMD benefited the most. It showed migration time saving up to 10 % (see Figure 4.7 (b)).

The combination of zeroing enabled and compression enabled, provided the least migration time for all application. 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 (see Figure 4.8 (b)).

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 contrary, mpiblast showed a minor improvement compared to PhyloBayes and NAMD. We relate this to the amount of partial zero page injected by the zeroing library. This will be studied more closely in Section 4.3.4.



(a)

(b)

Figure 4.7: Migration time and Migration time saving for NAMD with zeroing enabled or disabled when compression is enabled (dashed curves) or disabled (solid curves). The saving is with respect to the case when zeroing and compression are disabled.

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 by the savings that can be achieved with the given link speed.

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

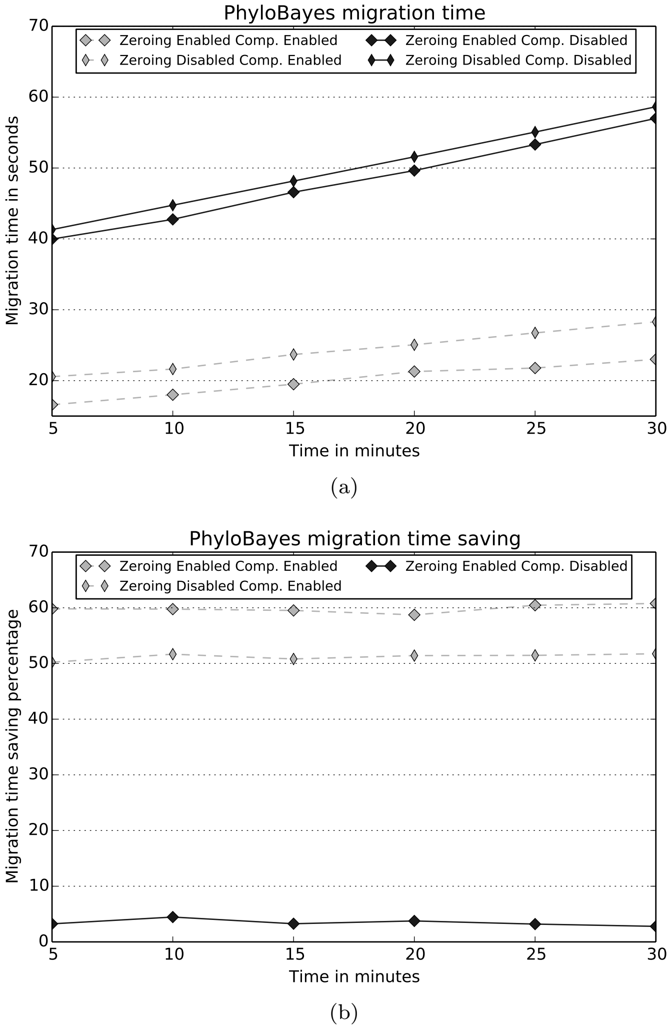
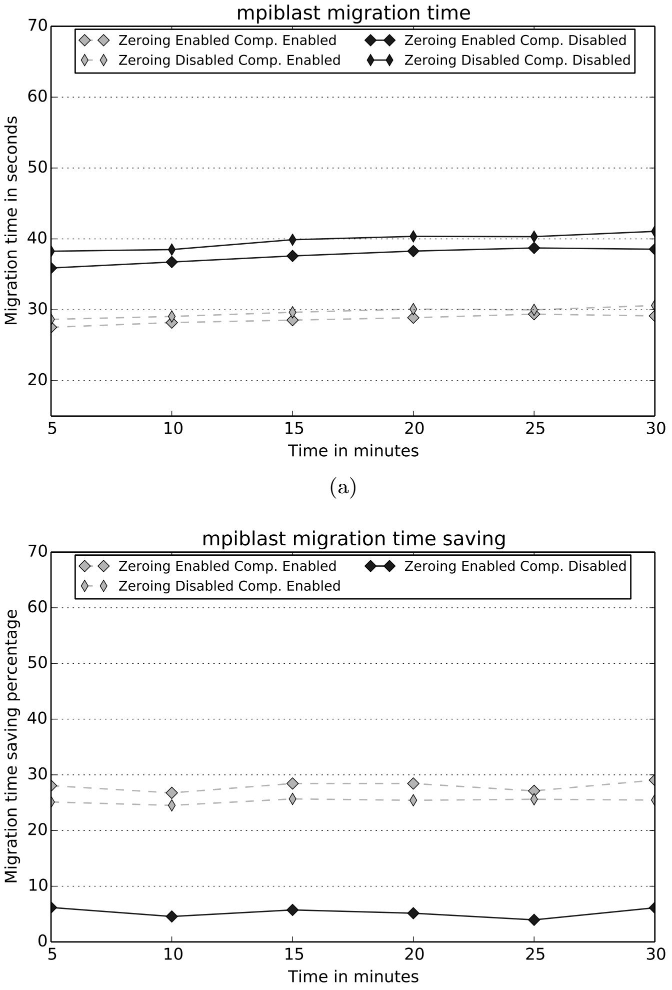


Figure 4.8: Migration time and Migration time saving for PhyloBayes with zeroing enabled or disabled when compression is enabled (dashed curves) or disabled (solid curves). The saving is with respect to the case when zeroing and compression are disabled.

4.3.4 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 4.3.3, 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 form



(a)

(b)

Figure 4.9: Migration time and Migration time saving for mpiblast with zeroing enabled or disabled when compression is enabled (dashed curves) or disabled (solid curves). The saving is with respect to the case when zeroing and compression are disabled.

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 4.3.3, 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

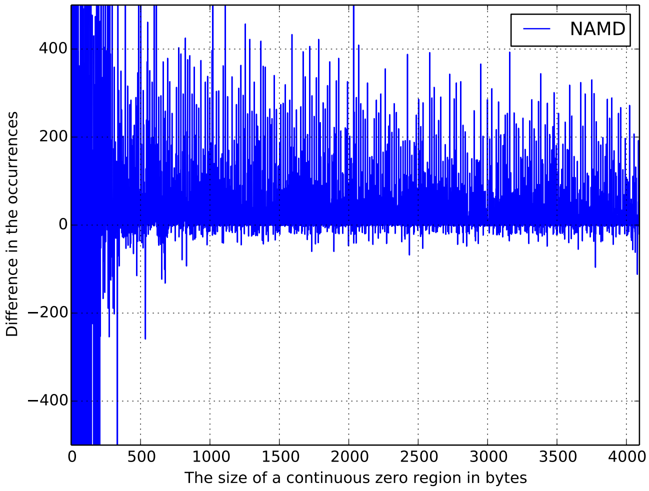


Figure 4.10: Difference in the number of occurrences of each zero region length for NAMD.

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. Figures 4.10 to 4.12 show the differences for NAMD, PhyloBayes and mpiblast, respectively. The results are clearly positive for NAMD, but not for mpiblast; for PhyloBayes it is ambiguous. Then we computed the cumulative function of the result(see Figure 4.13). The cumulative function accumulates the size of the differences in the partial zero pages. Figure 4.13 shows that the number of accumulated zeros increases for PhyloBayes and NAMD, but decreases for mpiblast. This explains, in 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 following questions: Why does zeroing 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 which matches our assumption 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.

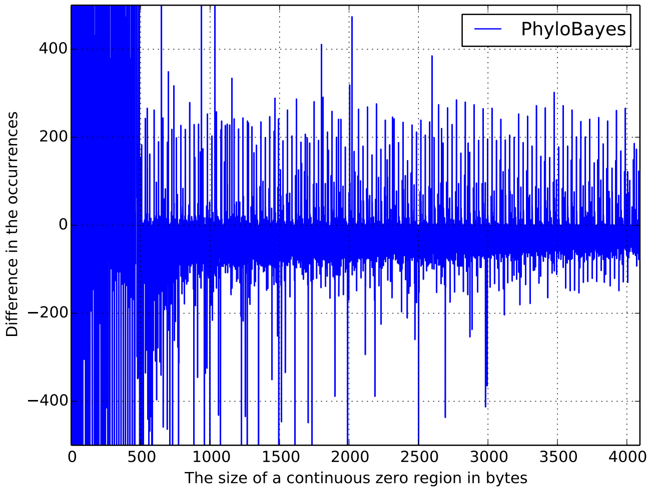


Figure 4.11: Difference in the number of occurrences of each zero region length for PhyloBayes.

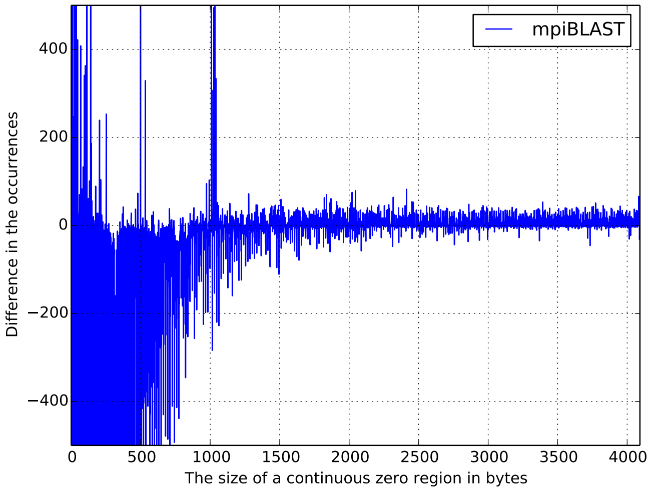


Figure 4.12: Difference in the number of occurrences of each zero region length for mpiblast.

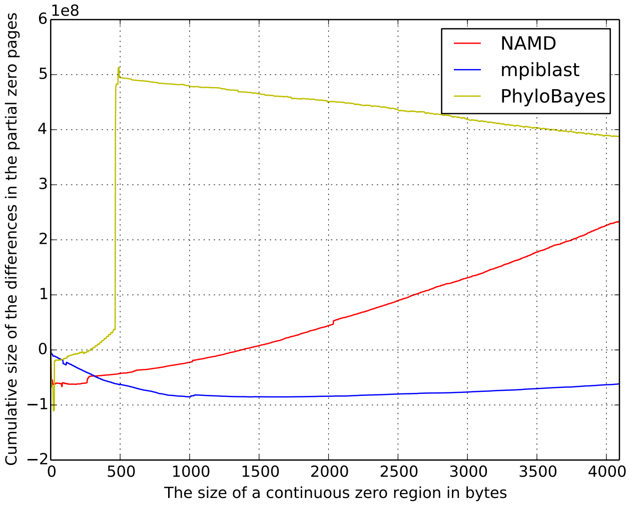


Figure 4.13: Cumulative function of the volume of the difference in the partial zero pages in bytes. The difference is between the two cases when zeroing enabled and disabled.

4.4 Generalization of the approach

In this section we discuss the current limitations of the memory management in the context of virtual machine migration/checkpointing. We reduce the size of virtual machines by writing zeros in freed memory regions, which benefits zero-page detection as well as compression schemes. However, freed memory regions that have been filled with zeros might be reallocated and reused by the application. For example, this is 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 4.14 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()*, and *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 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 create a new one. Of course, this should only be done when the application running in the VM terminate.

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 avoid unnecessary zero writes. The glibc library

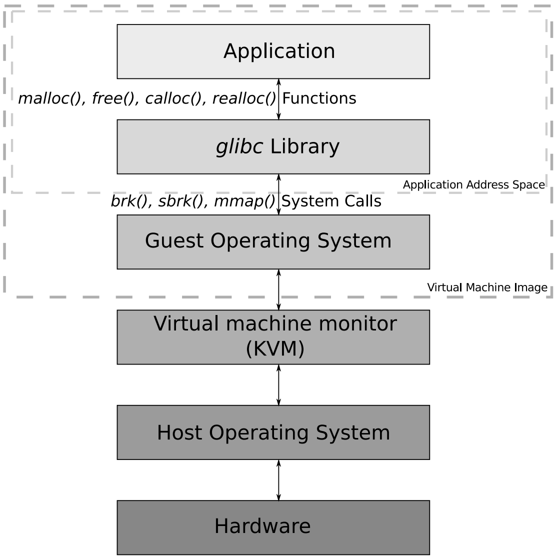

Application
malloc(), free(), calloc(), realloc() Functions
glibc Library
Application Address Space
brk(), sbrk(), mmap() System Calls
Guest Operating System
Virtual Machine Image
Virtual machine monitor (KVM)
Host Operating System
Hardware

Figure 4.14: Memory allocation stack. The application accesses memory using the *glibc* library which interacts with the guest operating system using system calls. The *glibc* library is a shared library in the application address space. The application and the guest operating system are included in the VM image.

needs to be modified to react upon migration/checkpoint requests (similar to the approach for the migration of MPI applications in [PCL+16]). In doing so, it would place zeros into freed memory regions before a migration/checkpoint operation takes place. 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 of mpiblast.

Instead of zeroing the memory regions, their addresses can also be recorded and skipped during migration/checkpointing. 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.

4.5 Conclusion

In this chapter, we have shown the limitations of the current memory management in the context of virtual machine migration/checkpointing and have presented a prototype, in order 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 94 % with compression. This reduces the storage required for checkpoints.

In Section 4.4, 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.

Chapter 5

Deduplication Potential of HPC Applications' Checkpoints

The desire for more computing power introduces new issues. The increasing number of nodes in clusters decreases the *mean time between failures* (MTBF). While the MTBF was in the order of days [IMB+12], it will further decrease for exascale systems [DBM+11, BBC+08, GCR+07]. Therefore, HPC programmers must consider that their (long running) jobs will not finish without any hardware or software failure. As a consequence, several programs integrate checkpoint mechanisms [EAWJ02], where the application writes its state to the storage backend and resumes its computation from this state after a node failure. This checkpointing approach is called *application-level checkpointing.* However, for legacy codes an integration of these mechanisms can be difficult and often exceeds its benefits. Even for non-legacy programs, the integration of checkpointing mechanisms is time consuming. Alternatively, *system-level checkpointing* can be applied, where an external tool checkpoints the running application without any assistance from the researcher.

Checkpointing leads to a new issue: most approaches do not scale well with the number of nodes. Creating checkpoints for many processes puts high pressure on the storage backend and on the network. Furthermore, the energy costs of moving data will exceed the costs of local computation in exascale environments [RD14]. A possible solution to increase checkpointing scalability is to apply data deduplication. Data deduplication systems have been introduced to reduce the amount of stored data on disk-based backup systems and have made these economically applicable. Most of these systems divide the data into chunks, compute chunk fingerprints using a cryptographic hash function like SHA-1, and identify redundant chunks by comparing the chunks' fingerprints [MKB13, XJFH11, BELL09, LEB+09, ZLP08]. Besides backup, primary storage also provides deduplication potential. Meister et al. showed a *huge potential for data deduplication in HPC storage systems that is not facilitated by today's HPC file systems* [MKB+12]. Nicolae and Kulkarni et al. applied the deduplication approach to the checkpointing use case and discussed systems to reduce the I/O load during checkpointing [Nic13, KMI+12].

However, there is little knowledge about the general deduplication potential of HPC applications' checkpoints. The study of a broad spectrum of applications presented in this chapter give new insights into the applicability of checkpoint deduplication in different application areas. The coupling of checkpointing and deduplication for applications, which have a high deduplication rate, can definitely help to improve checkpointing scalability. Nevertheless, if an application does not have enough redundancy, the deduplication process can decrease the overall checkpointing performance. In addition, an improperly configured deduplication process can waste deduplication potential. Our experiments show, for example, that choosing the wrong chunking process alone can alter the volume of the data after deduplication by 10%.

In this study, we investigate the deduplication behavior of a wide range of HPC applications. For each application we show its deduplication potential for different deduplication configurations. Since not all applications provide built-in checkpointing, we use DMTCP for system-level checkpointing [AAC09]. Using this type of checkpointing, we show that there is a high potential for saving data sent to disk and for increasing checkpointing performance. Furthermore, we show where redundancies occur and how they can be exploited best.

5.1 Related Work

Reducing an application's checkpoint size is a potential source of improving the performance and the scalability of a checkpointing process. In addition to storage reduction, a smaller checkpoint means smaller dumping time, smaller bandwidth consumption, and less I/Os.

The simplest checkpoint approach is a memory core dump of the computation status, which is called system-level checkpointing. It can be implemented in the kernel space in case of BLCR (Berkeley Lab Checkpoint/Restart) [Due03] or in the user space in case of DMTCP (Distributed MultiThreaded CheckPointing) [AAC09]. The advantage of system-level checkpointing is that it is transparent to the application and the checkpoint can be taken at arbitrary points. However, those tools produce relatively large checkpoints, as they include data that is not required for restarting the computation.

Application-level checkpointing was introduced to provide smaller-sized checkpoints by adding more complexity. The programmer exploits the knowledge of the structure and the behavior of a given application to find the useful data structures that should be included in the checkpoint and encodes this knowledge in the application's source code.

Library and compiler support have been proposed to simplify the complexity of application-level checkpointing. The `Libckpt` library provides transparent checkpoint/restart, but it requires user directives to mark the checkpoints' locations and data [PBKL95]. Bronevetsky et al. provide a source to source compiler tool that can automatically instrument the code to save and restore its own status. The tool coordinates checkpoints and restarts for parallel OpenMP [BPS06,BMP$^+$04] and MPI programs [SBF$^+$04,BMPS03b,BMPS03a].

Compression techniques [IAB$^+$12] and incremental checkpointing [NC13] were introduced to decrease the checkpoint size. They are applied to the raw checkpoint data. Incremental checkpointing only save the differences between checkpoints instead of saving the complete checkpoints.

Differences between checkpoints also can be created by tracking writes to memory pages. Only these pages are included in the checkpoint [VMHR11, GSjP05]. The disadvantage of this approach is that it requires kernel-level support.

Fingerprinting-based deduplication is also used to find differences between checkpoints. The data is partitioned into non-overlapping data blocks (chunks). A cryptographic hash function is used to obtain a unique identifier (hash value) for each chunk. The chunks' hash values are compared against each other so that repeated chunks are only written once.

Data chunks can have a fixed size (static chunking, SC) or variable size (content-defined chunking, CDC). SC is simple and fast, but it fails to detect duplicates in near identical, but shifted data streams. Cores et al. combine SC hashing-based deduplication with incremental application-level checkpointing [CRMG12]. CDC finds the boundaries of chunks based on their content and therefore overcomes the data shifting problem [MCM01]. The chunks generated by CDC have variable chunk sizes within an upper and lower limit [MCM01].

Kulkarni et al. [KMI$^+$12] describe a deduplication-based file system to checkpoint HPC applications. The system is evaluated using checkpoint dumps of five proxy implementations of real HPC applications and three dumps of full applications. While the experiments show a great variation in the deduplication ratio, there is no analysis of how precisely the proxies reflect the deduplication potential of the respective full applications.

5.2 Deduplication of Checkpoints

A system designer faces several challenges and design decisions when he creates a deduplication system for HPC checkpoints. The overall goal is to perform deduplication fast while detecting as much redundancy as possible and keeping the resource requirements low. One major choice is the chunking method and the chunk size. Content-defined chunking can allow better redundancy detection than fixed-size chunking, but induces a higher computation overhead.

The chunk size is a vital parameter because it influences the quality of redundancy detection and the number of chunks and thus the processing time: the smaller the chunk size, the more fine-grained the detection and the longer the processing. We will investigate these correlations in Section 5.4.1.

Another disadvantage of small chunks is that the size of critical data structures grows with the number of chunks. For example, each deduplication system holds an index mapping chunks to the storage location of their raw data. The size of an index entry typically ranges from 24 B to 32 B, including hash value, storage location, and counters and pointers for the index implementation; therefore, each stored terabyte of unique checkpoint data requires 4 GB of extra memory if we assume 20 B SHA1 hashes and 8 KB chunks, which allows the deduplication system to hold the full index in memory. Consequently, no disk I/Os are required in the deduplication process except for writing new chunks to disk.

Finally, the system designer must consider scaling properties. The probably best scaling approach is to let each compute node perform its own deduplication and store raw chunk data on local storage. However, all checkpoints for that node would be lost in case of a hardware failure. On the other side, a single deduplication instance can easily become a performance bottleneck in the presence of thousands of processes and nodes performing checkpoints. Therefore, it is advisable to replicate chunk data to other nodes, which reduces the savings achieved by the deduplication process.

5.3 Methodology

In this section, we describe the applications, checkpointing and data deduplication tools used in our study. We also show how we perform the checkpoint generations and the data deduplication chunking scheme that we used.

5.3.1 Applications

We evaluated applications from different scientific areas like physics, chemistry, material science, meteorology, and biology to understand the deduplication behavior across a broad range of applications and user bases. Our study used the applications described in Section 4.2.3 in addition to the following applications.

pBWA [PLQL12] is an MPI implementation of BWA [LD09], a popular software package for mapping low-divergent sequences against a large reference genome, such as the human genome. The data distribution among the available processors consists mainly of two parts: index distribution and sequences distribution. First an index file is generated as BWA. This file is read by a master process and then broadcast to all available processors. After that pBWA distributes the sequences and computes the alignments on each process. In our experiments, we let pBWA compute the alignments of 38 million short reads against a human genome.

bowtie computes alignments of DNA sequences similar to pBWA and mpiblast but is specialized in short read sequences. It *conducts a quality-aware, greedy, randomized, depth-first search through the space of possible alignments* [LTPS09]. bowtie itself is a serial program, which is often used in parallel fashion. An MPI-based tool called pMap [Pma] is used to parallelize the execution of bowtie. The parallelization performed by pMap is conceptually similar to the one of pBWA. The index file, which is associated with the reference genome, is replicated on every processor and the reads contained in the input file are distributed among the processors. In our experiments, we let bowtie compute the alignments of 38 million short reads against a human genome.

ray [BLC10] is a parallel *de novo genome assembler* for next-generation sequencing data. It takes a FASTA file as input. In our experiments, the input FASTA file contained 9,650,581 sequences. The sequences were distributed among the available MPI ranks and each rank got an equal number of sequences assigned to it.

Espresso++ [HBL$^+$13] is a software framework targeted at the simulation of soft matter systems. We ran a simulation that used an adaptive resolution scheme (AdResS) in which regions of the simulation box had different levels of chemical details. It took an input file which contained a list of particles that make up the system under study. The cubic simulation box in our evaluation contained 20,004 particles for which initial position coordinates, velocities, and other information were provided. Espresso++ uses domain decomposition to distribute the data among the available processors.

nwchem [VBG$^+$10] is a software used for computational chemistry simulations. The chosen benchmark applies the *density functional theory* (DFT) on the C_{240} molecular system. The input file contained the coordinates of all 240 carbon atoms as well as the steps to be performed in the simulation. nwchem uses domain decomposition to distribute the data among the available processors.

CP2K [HISV14] is an open source framework written in Fortran that enables users to perform molecular simulations based on the *density functional theory* (DFT). In our experiments, we ran a molecular dynamics simulation of a system of 128 water molecules constituting a periodic volume box of dimensions (15.6404, 15.6404, 15.6404) at the pressure of 1 bar and at the temperature of 300 K. The simulation produced positions, velocities and forces at each time step for every atom.

Quantum ESPRESSO (QE) [GBB$^+$09] is a set of codes written in Fortran used to calculate electronic structures and to model materials. In our experiments, we used a code called CP which performs a variable-cell Car-Parrinello molecular dynamics simulation. The system under study was a set of 512 water molecules. In particular, the system was a Triclinic lattice for which the crystallographic constants were specified. Among other things, the input file contained the atomic species, their characteristics, and the positions in the crystal of all the $1,536$ atoms (512 O atoms $+ 2 * 512$ H atoms). QE implements parallelization at different levels to use the available processors for the simulation.

eulag [PSW08] (**Eu**lerian/semi-**Lag**rangian fluid solver) is a numerical solver for geophysical flows written in Fortran. In our experiments, we ran a Large-Eddy simulation using the Smagorinsky subgrid-scale model and used PnetCDF [LkLC$^+$03] as a parallel I/O library. The model grid size was (160, 96, 60) and domain grid decomposition was used to map the model grid to the available processors.

echam [SGE$^+$13] is a climate model which simulates atmospheric general circulation. In our experiments, we used ECHAM5, which was written in Fortran, and simulated weather conditions starting from January 1998. The dataset consisted of many files containing the initial state of the weather grid. The data was distributed via domain grid decomposition to the processors.

openfoam [JJT07] is a toolbox of numerical solvers for computational fluid dynamics problems. Our tests followed a standard user workflow including preprocessing steps before calling the final numerical solver. Preprocessing consisted of domain decomposition (*decomposePar*), allocating the subdomains to processors and checking the validity of the mesh produced.

As in a standard user workflow, we performed some pre-processing steps before calling the final numerical solver. First we used the decomposePar utility, which performed domain decomposition by splitting mesh and fields into sub-domains which are allocated to the different processors. We then checked the validity of the mesh produced. The mesh involved in the simulation had the following main parameters: nPoints=$528,732$, nCells=$498,350$, nFaces=$1,524,519$, nInternalFaces=$1,465,581$. Finally, we ran the icoFoam solver, which is a transient solver for incompressible, laminar flow of Newtonian fluids.

5.3.2 Checkpoint Generation

In this study, we use the DMTCP tool [AAC09] to generate the applications' checkpoints. We chose DMTCP because it does not require root privileges and is independent from the kernel version. DMTCP provides checkpointing at system-level. These checkpoints can be compressed during creation. We disabled this feature for our analysis since a compression before the redundancy detection of the deduplication destroys the latter. Deduplication systems typically use compression after the chunk identification when they write the raw chunk data to disk. Unlike application-level checkpointing, system-level checkpointing is transparent to the applications and can be done at arbitrary points. Tools for system-level checkpointing also allow the checkpointing of legacy codes, which cannot be modified. On the other hand, system-level checkpoints generate relatively large checkpoints in comparison to application-level checkpoints. As system-level checkpointing includes more redundant data, it is expected that these process images also have a higher deduplication potential than application-level checkpoints. However, system-level checkpointing offers more freedom in choosing the checkpoints' locations transparently to the application.

DMTCP generates one checkpoint image for each MPI process. The image is composed of a global header section, a header for each contiguous memory area (contains address range, permissions, etc.), and the data section (memory pages) for the different contiguous memory areas. The header section consists of 4 KB or one memory page. The first memory address of a continuous memory block is always a multiple of 4,096. Therefore, all checkpoint images are page-aligned.

We generate checkpoints every ten minutes. Almost all applications perform their computations for two hours. Only bowtie (after 50 minutes) and pBWA (after 110 minutes) finished earlier. The computations are distributed among 64 processes for all applications. The checkpointing period, the total applications' execution time, and the number of processes are chosen based on the possibility of having comparable characteristics between a large set of applications. However, we vary the number of used processes in Section 5.4.3. Table 5.1 shows the different sizes of the checkpoints.

5.3.3 Deduplication

We analyzed each checkpoint with the FS-C deduplication tool suite [Mei11], which has already been applied in several deduplication studies [MKB+12, KMBS15]. We used fixed-sized chunking and content-defined chunking (CDC) as chunking methods. For CDC, the suite uses Rabin's fingerprinting [Rab81] to determine chunk boundaries and computes the SHA1 chunk fingerprints. For each checkpoint, we generated traces with an (average) chunk size of 4, 8, 16, and 32 KB.

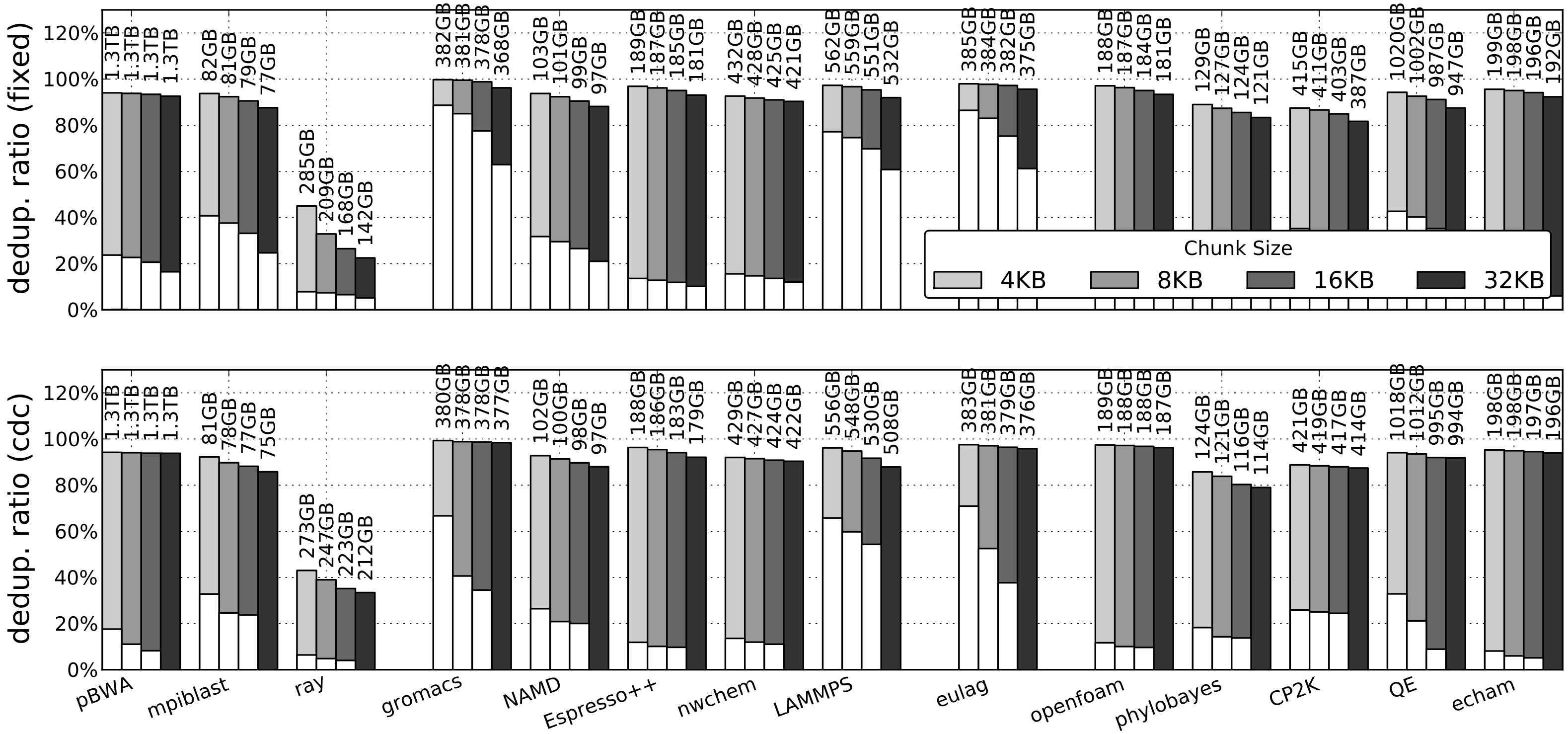


Figure 5.1: Deduplication ratio of all applications. The upper figure shows the ratios for fixed-size chunking, the lower figure shows the ratios for content defined chunking. The values above bars show the absolute volume of the redundant chunks. The white bars indicate the relative volume of the zero chunk.

Table 5.1: Checkpoint size statistics for all applications, each running on 64 processes.

Application	avg	sum	min	25%	75%	max
pBWA	132GB	1.4TB	35GB	52GB	184GB	185GB
mpiblast	33GB	405GB	33GB	33GB	33GB	33GB
ray	75GB	902GB	37GB	70GB	89GB	93GB
bowtie	94GB	470GB	1.2GB	65GB	134GB	175GB
gromacs	34GB	418GB	34GB	34GB	34GB	34GB
NAMD	10GB	120GB	10GB	10GB	10GB	10GB
Espresso++	17GB	213GB	13GB	18GB	18GB	18GB
nwchem	42GB	511GB	29GB	43GB	43GB	43GB
LAMMPS	52GB	631GB	52GB	52GB	52GB	52GB
eulag	35GB	428GB	35GB	35GB	35GB	35GB
openfoam	17GB	213GB	3.2GB	19GB	19GB	19GB
phylobayes	39GB	473GB	39GB	39GB	39GB	39GB
CP2K	43GB	518GB	37GB	43GB	43GB	43GB
QE	99GB	1.2TB	74GB	88GB	109GB	109GB
echam	18GB	227GB	18GB	18GB	18GB	18GB

Memory deduplication usually uses fixed-size chunking with a chunk size equal to the physical memory page size [BWSS12, MFR+13]. We generate the same page alignment for fixed sized chunking and 4 KB chunks since the embedded process images are page-aligned.

5.4 Evaluation

The deduplication potential of data contains different aspects. We first show general deduplication properties of the applications, i.e., the general deduplication ratio. Then, we analysis the main source of redundancy (Section 5.4.2). Finally, in Section 5.4.3, we investigate the deduplication behavior of the applications' checkpoints with increasing the number of the computing processes.

All experiments were run on nodes of the HPC cluster *Mogon* of the Johannes Gutenberg University Mainz. Each of the nodes is equipped with four AMD Opteron 6272 processors and at least 128 GB RAM. Each processor has 16 cores. All nodes have access to a local scratch and a global GPFS file system.

5.4.1 General Deduplication

For each application, we ran a computation that was distributed among 64 cores. For each computation, we created checkpoints every 10 minutes for a total computation time of two hours. Next, we deduplicated all checkpoints and computed the overall deduplication ratio, which is defined as $1 - \frac{\text{stored capacity}}{\text{total capacity}} = \frac{\text{redundant capacity}}{\text{total capacity}}$. A deduplication ratio of 80% denotes that 80% of the data could be removed by a deduplication system and 20% of the original data would be stored. Figure 5.1 shows the deduplication ratios for all applications for fixed-size chunking (upper subfigure), content-defined chunking (lower subfigure), and

Table 5.2: Deduplication ratio and zero chunk ratio for all applications. The values for *single* denote ratios for the single checkpoint. All values were collected for a 64 processes run and fixed-sized chunking with a chunk size of 4 KB.

Application	single		
	20 min	60 min	120 min
pBWA	91% (17%)	92% (17%)	
mpiblast	99% (92%)	99% (92%)	99% (91%)
ray	97% (77%)	39% (34%)	37% (32%)
bowtie	74% (23%)		
gromacs	99% (88%)	99% (88%)	99% (88%)
NAMD	81% (31%)	81% (31%)	81% (31%)
Espresso++	79% (13%)	79% (13%)	79% (12%)
nwchem	66% (12%)	89% (12%)	89% (12%)
LAMMPS	97% (77%)	97% (77%)	97% (77%)
eulag	97% (88%)	97% (85%)	97% (84%)
openfoam	89% (13%)	89% (13%)	89% (13%)
phylobayes	95% (79%)	95% (79%)	95% (78%)
CP2K	81% (32%)	81% (32%)	80% (32%)
QE	65% (55%)	57% (38%)	57% (38%)
echam	93% (10%)	92% (10%)	92% (10%)

different (average) chunk sizes. The values above the bars indicate the absolute total volume of the redundant data[1].

Except for ray, all applications show a deduplication ratio of more than 84%. Smaller chunks enable better redundancy detection. For fixed-size chunking, the maximum difference between 4 KB and 32 KB chunks for the same application is 9.8%. For CDC, the difference is 8.3%.

The white bars show the ratio of the zero chunk, i. e., the chunk that contains only zeroes. This ratio is defined as $\frac{\text{zero chunk capacity}}{\text{total capacity}}$. In their HPC study, Meister et al. found that between 3.1% and 24.3% of their HPC data at best consisted of zero chunks [MKB$^+$12]. In our case, the zero chunk is the most used chunk and is the main source of redundant data for every application and chunk size, except CDC with an average chunk size of 32 KB. Note that the zero chunk has the property of always having the maximum chunk size, if content-defined chunking is used. In our setting, this size is four times the (average) chunk size, i. e., a zero chunk for CDC, 16 KB ranges over 64 KB = 16 memory pages. Note that the zero chunk ratio for 16 KB and CDC is smaller than for 4 KB and fixed-size chunking because CDC does not preserve page alignment.

Next, we investigated how the zero chunk ratio and the deduplication potential vary over time. We determined the deduplication ratio and the zero chunk ratio for *single* checkpoints after 20min, 60min, and 120min. These values are shown in Table 5.2. The parenthesized values denote the zero chunk ratio. The zero chunk ratio is constant for 13 applications. Only for ray and QE we see a significant drop from 77% to 34% and 55% to 38%, respectively.

[1] Note that we ignored the last checkpoint in the figure so that pBWA could be included. Therefore, the values for the saved amount of data should not be compared to Table 5.1 which displays values for all available checkpoints.

Table 5.3: Comparison of application-level and system-level checkpoint sizes for a subset of the tested applications.

Application	sys-lvl (+dedup)	app-lvl (+dedup)	$\frac{\text{sys-lvl+dedup}}{\text{app-lvl+dedup}}$
NAMD	10 GB (559 MB)	15 MB (15 MB)	37
gromacs	34 GB (83 MB)	65 KB (65 KB)	1328
LAMMPS	52 GB (1.4 GB)	1.5 MB (1.5 MB)	955
openfoam	17 GB (513 MB)	56 MB (55.9 MB)	12
CP2K	43 GB (5.4 GB)	21 MB (21 MB)	263
ray	75 GB (28 GB)	30 GB (29.6 GB)	0.93

Overall, the zero chunk is the biggest single source of deduplication such that a zero chunk deduplication alone saves at least 10% of the checkpoint data.

Application-Level vs. System-Level Checkpointing Many HPC applications provide their own checkpointing. These checkpoints can be significantly smaller because programmers know best which data is necessary to represent a computation's state. Table 5.3 compares the average checkpoint sizes of system-level checkpointing and and application-level checkpointing for a subset of the test applications. The last column in Table 5.3 displays the factors by which system-level and application-level checkpoints differ after deduplication.

For almost all applications, the average system-level checkpoint size is several orders of magnitude larger than the application-level checkpoint. However, the storage requirements for system-level checkpoints decrease significantly. In case of ray, the new storage capacity requirements is even lower than for the application-level checkpoint. Therefore, deduplicating system-level checkpoints can outperform application-level checkpointing.

There is a high deduplication potential in every application. The difference between fixed-size and content-defined chunking is small. The zero chunk is the dominant source of redundancy.

5.4.2 Stability of the Input

In this section, we take a closer look at the source of the redundancy, besides the zero chunk. In detail, we investigate whether the redundancy is defined by the input data or by data generated during the computations.

As input data, we define the content of the heap at the point the application closes the input file(s) the last time. This follows the intuition that the full state of the process up to that point contains data from initialization and the input but is untouched by the core computation.

For this, we ran single-process computations of QE, pBWA, NAMD, and gromacs and paused the computation after the last close call of the input file(s), as well as after every 10 minutes of computation afterwards. We choose these applications because of their different memory footprints. At each interruption, we created a checkpoint by copying the full process image via the `/proc` file system. We are interested in the part of the image, which includes the input data and will contain most of the new generated data over time. Therefore, we extracted the heap part of the image and removed the data of shared libraries and the application's object code because this data is static or defined by external

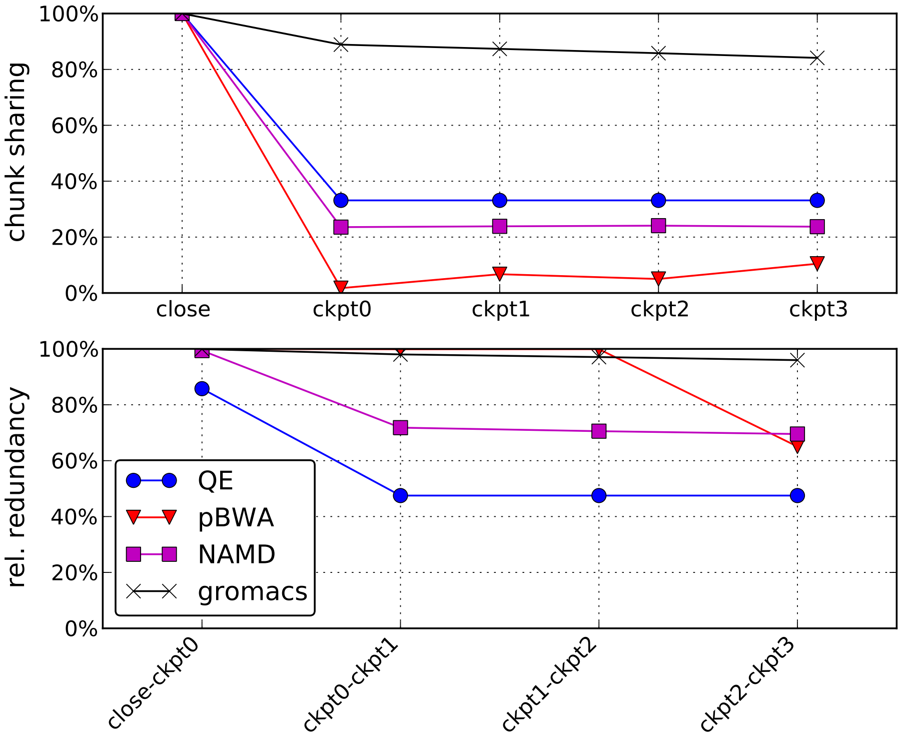


Figure 5.2: Upper plot: relative volume of the input data in the following checkpoints. Lower plot: the input data's share of the redundancy in the (windowed) deduplication.

libraries. Next, we chunked and fingerprinted the heap with fixed-size chunking with a chunk size of 4 KB. Hence, every chunk corresponds to a memory page. We refer to the checkpoint that only includes the input data as *close-checkpoint*.

Next, we computed the input data's share of the later checkpoints by checking each chunk of a later checkpoint whether it already existed in the close-checkpoint. The upper plot of Figure 5.2 shows the resulting relative volumes (chunk sharing). All shares are 100% for the close-checkpoint since a checkpoint shares every chunk with itself. As can be seen, the shares of NAMD and QE are near constant at 24% and 38%, respectively, while the share of gromacs decreases from 89% to 84%. pBWA input data's share increases from 2% to 10%. This is counterintuitive because applications generate data, which increase the checkpoint sizes and, therefore, should reduce the input data's share. A closer look showed that pBWA's checkpoints do not shrink; therefore, pBWA generates the share increase by copying parts of the input data internally.

However, the shared data has a big impact on the deduplication ratio. Next, we determined the input data's share of all redundant chunks. For this, we took each two consecutive checkpoints, determined the redundant chunks, and checked for each one whether it already existed in the input data. We did not use single checkpoints because this would ignore inter-checkpoint redundancy. A share value of 80% denoted that 80% of the redundant chunks also existed in the input and, therefore, that 80% of the redundancy is based on the input. The lower plot in Figure 5.2 shows the shares for the applications. In general, more than 48% of the redundancy is based on the input data. For all applications, the share decreases over time as they generate new data which is redundant among the checkpoints.

Besides the zero chunk, more redundancy originates from input data than from data generated during the computations.

5.4.3 Scaling Effects

In all experiments mentioned above, the number of processes was set to 64. Now we vary the number of processes and consequently also the distribution of input data and computation over the processes.

For this analysis, we selected applications from different domains, namely mpiblast, NAMD, and phylobayes, and also added ray because of its relatively low deduplication potential. We applied fixed-size chunking with a chunk size of 4 KB. For better scalability support, we changed the DMTCP version (2.4.0 instead of 2.3.1) and the MPI library (Open MPI 1.8.1 instead of MPICH 3.1) compared to the previous experiments. For this reason, the deduplication ratio in Figure 5.3 should not be compared to Table 5.2. Figure 5.3 shows the accumulated deduplication ratio and the zero chunk ratio for a different number of processes.

The deduplication ratio of all tested applications but ray increases with the number of processes until 64 processes are reached. Beyond this number – which also marks the number of cores per node in our test system – we see three different behaviors: the ratios of mpiblast and phylobayes decrease, the ratio of NAMD increases after an initial drop, and the ratio of ray remains at the same level after an initial drop.

The deduplication potential is high, independent of the number of processes. The zero chunk is the dominant source of redundancy, even for a large number

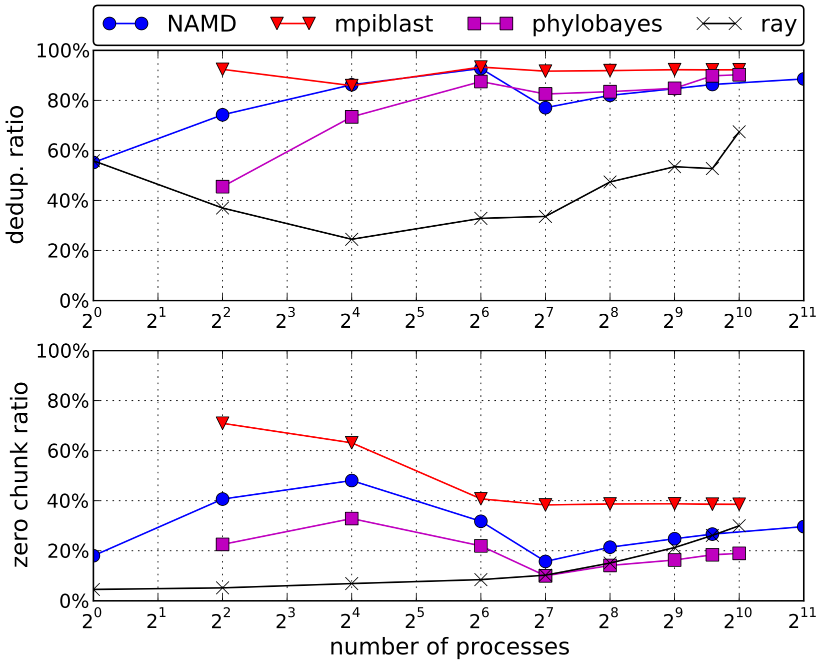


Figure 5.3: Upper plot: deduplication ratio for a different number of processes. Lower plot: zero chunk ratio for the same experiments.

of processes.

5.5 Conclusion

In this work, we studied the deduplication potential of system-level checkpoints for a broad range of HPC applications. We can summarize that all applications show significant savings potential independent of their domain and their underlying computation model; the potential ranges from 37% to 99%. The results suggest that some applications sustain a high potential for a larger number of nodes. The evaluation further shows that even rather simple deduplication approaches can eliminate most of the redundant data. For example, removing the most frequent chunk, the zero chunk, reduces the checkpoint data by 10-92%.

In our experiments, deduplication could reduce the storage requirements of system-level checkpointing by several orders of magnitude, but application-level checkpointing, with one exception, still required at least one order of magnitude less storage space.

Due to the given page structure, fixed-size chunking is a natural choice. In contrast to traditional deduplication systems, content-defined chunking does not detect redundancy better.

We hope that this study provides a solid foundation for the design of future checkpoint deduplication systems. In the next Section, we show our vision for this system.

5.6 Scalable Checkpoint for HPC

A checkpoint approach is scalable at a certain system size if the overhead of checkpointing is affordable at that size, i. e., can be neglected. Decreasing the checkpoint overhead is key to achieve better scalability. A smaller checkpoint has less processing time and storage which means less overhead.

In this chapter, we have shown the potential of applying data deduplication on HPC applications checkpoints to reduce their sizes. Data deduplication exposes the redundancy between application processes checkpoints to reduce their storage size. Islam et al. explore the same redundancy to reduce the checkpoint size by relying on metadata semantic in the checkpoint files [IMB+12]. This is what they call data-aware checkpoint compression.

Our contribution, in this chapter, can be combined with other techniques in the literature, like multi-level checkpointing [MMBdS14, MBMS10], asynchronous checkpointing [MMdS12], and cooperative checkpointing [ORS06b, ORS06a] to design a better scalable checkpointing scheme. As an extension of our work, we propose the design of a new checkpointing system that is built on our contribution and make use of other checkpoint enhancement techniques.

Our proposed system is similar to Sant et al. which is a nonblocking checkpointing system that combines the benefits of asynchronous checkpointing, multi-level checkpointing and data staging [SMM+12]. However, we would like to enhance this system with checkpoint deduplication and cooperative checkpointing.

Our proposed system is shown in Figure 5.4. The system is composed of two node sets: computing node and staging-deduplication nodes. Applications execute on the computing nodes and perform multi-level checkpointing with the Scalable Checkpoint/Restart (SCR) library [MMBdS14, MBMS10]. The checkpoint levels include checkpointing to the local node memory, the neighbor node memory, and the Parallel File System (PFS).

Checkpointing to the PFS is performed through the staging-deduplication nodes. They read checkpoints from the computing nodes using Remote Direct Memory Access (RDMA). In this way, applications running on the computing nodes are not blocked by the checkpointing operations. Every staging-deduplication node is responsible for collecting checkpoints from a group of computing nodes.

Collected checkpoints on the staging-deduplication nodes are deduplicated together, and then they are sent asynchronously to the PFS. Applying data deduplication on the checkpoints before sending them to the PFS, can significantly reduce the checkpoint size. We also recommend that the deduplicated checkpoints should belong to the same application processes to have a better chance of reducing the checkpoint size.

Asynchronous sending data to PFS can be tuned to decrease the effective load on the PFS [MMdS12]. Cooperative checkpointing can also be employed in this checkpointing scheme by giving the system the possibility to accept or deny a checkpoint operation based on the system resources' status, expected failures, scheduling strategies, Quality of service guarantees, etc. [ORS06b, ORS06a].

Finally, we hope that our vision for a scalable checkpoint system inspires system developer to design an optimal system.

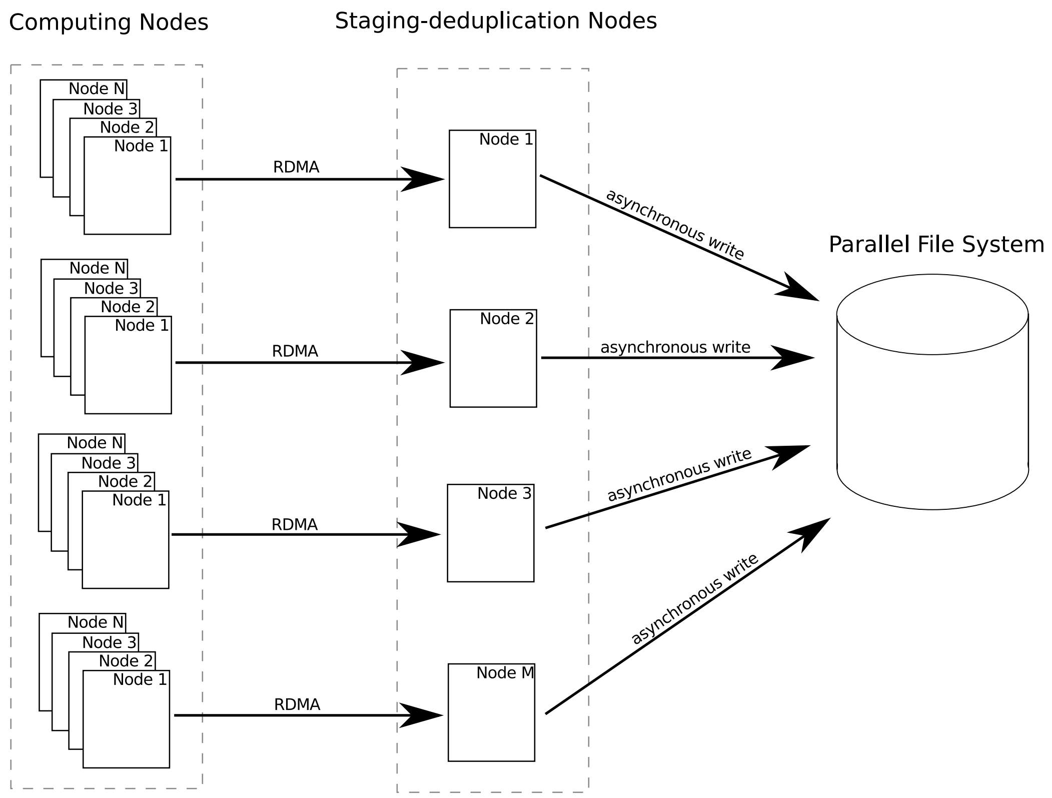


Figure 5.4: A block diagram of our proposed scalable checkpointing system. The system has two sets of nodes: computing node and staging-deduplication nodes. The arrows show the direction of flow of the checkpoints. Staging-deduplication nodes read checkpoints from the computing nodes using RDMA. Every staging node is a response of collecting checkpoints from a set of computing nodes. Data deduplication is applied on the collected checkpoints then asynchronously written to the PFS.

Chapter 6

Conclusions

Application checkpointing and migration are key solutions to enhance fault tolerance and to balance workloads between computing nodes in order to avoid resource congestion. However, applying them to a large exascale environment introduces new challenges. In this thesis, we addressed parts of these challenges.

In Chapter 3, we consider checkpointing for load balancing between different resources on a heterogeneous node. It is connected to context switching between the host's memory and the accelerator's memory. We presented the tool collection *ConSerner* that automatically identifies, gathers, and serializes the context of an original CPU kernel and migrates it to an accelerator's memory where an accelerator kernel is executed with this data. The overhead introduced by our tool could be compensated by exploiting the advantages of the accelerator over the ordinary CPU. In our contributions to the state of the art, we showed by integrating *ConSerner* in a resource-aware scheduler, we were able to balance workload between computing cores and accelerators in a heterogeneous node. This way, we are able to use all the resources on the heterogeneous node more efficiently.

In Chapter 4, we proposed an approach to accelerate VM migration and reduce VM checkpoint size in the HPC context by a reduction of the VM image size. This approach overwrites unused data with zeros, so that zero-page detection and compression schemes work more efficiently. In our evaluation, we demonstrated that our approach is suitable for HPC environments by testing it with parallel applications encapsulated in virtual machines. We showed that 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 also showed that our approach reduced the checkpoint size of our tested applications by up to 9 % without compression and up to 94 % with compression.

Finally, in Chapter 5, we considered the challenges associated with checkpointing. We studied the deduplication potential of system-level checkpoints for a broad range of HPC applications. We could summarize that all tested applications have shown significant saving potential independent of their domain and their underlying computation model; the potential ranged from 37% to 99%. The results revealed that some applications sustain a high potential even for a larger number of nodes.

Reducing the checkpoint storage requirements helps to improve scalability. We hope that this study provides a solid foundation for the design of future deduplication systems. In this chapter, we have also proposed our vision for a scalable checkpointing system.

Finally, these contributions to the state of the art are just a few steps on the road to the exascale era, and there are more steps to go. We hope that this work will help other researchers to add more contributions. We end this thesis with a couple of quotes. Humanity strive for perfection has no end and will never be reached. Every beginning has an end, and every end is just a new beginning. This is the end of this thesis.

Appendix A

ConSerner

ConSerner is publicly available under `https://version.zdv.Uni-Mainz.DE/anonscm/git/conserner/conserner.git`

Inside conserner folder you find the following:

conserner_lib contains *ConSerner* dynamic library. Use the makefile to compiler the library. You may need to modify the makefile to fit to your system. You also need CUDA installed on your system.

llvm3.5_modified_conserner_ICDCS2014 contains the source code for our modified LLVM compiler with clang. It is based on LLVM version 3.5. *ConSerner* compiler plugin is already integrated in this modified copy. Follow the instruction here `http://llvm.org/docs/GettingStarted.html` to compile.

organizedSamples contains *ConSerner* sample examples. You may need to modify the makefile to fit on your system.

Appendix B

Zeroing Preload Library

The zeroing preload library that we have developed in Chapter 4 is publicly available under `https://version.zdv.Uni-Mainz.DE/anonscm/git/memory-zeroing/memory-zeroing.git`.

Acronyms

BLCR Berkley Lab Checkpoint / Restart for LINUX. 8, 38

CPU Central Processing Unit. 12, 34, 36, 96

CR Checkpoint / Restart. 63

DMTCP Distributed MultiThreaded Checkpointing. 9

DSP Digital Signal Processing. 36

FLOPS Floating Operation Per Second. 1

FPGA Field Programmable Gate Array. 36

GPU Graphics Processing Unit. 28, 29, 36, 47, 58

HDFS Hadoop Distributed Filesystem. 21

HPC High Performance Computing. i, 1, 3, 5, 6, 12, 18, 21, 28, 29, 61–64, 67, 72, 79, 81–83, 88–90, 96, 102, 104

I/O Input Output. 2

KVM Kernel-based Virtual Machine. 5, 61, 62, 64, 73

MPI Message Passing Interface. 8, 9, 16, 22

MTBF Mean Time Between Failures. i, 2, 6, 28

NUMA Non-uniform memory access. 72

PFS Parallel File System. 6, 12, 17–21, 28, 94, 95, 102

PID Process ID. 8

PSU Power Supply Unit. 28

QoS Quality of Service. 20

RAID redundant array of inexpensive disks. 16

RAM Random Access Memory. 16–18, 102

RDMA Remote Direct Memory Access. 18, 21, 94, 95

SCR Scalable Checkpoint/Restart. 18, 21, 94

SSD Solid State Drive. 16, 17, 102

VM Virtual Machine. 61, 62, 64, 65, 69, 72, 74–76, 78, 79, 96

VMM Virtual Machine Monitor. 33

List of Figures

List of Tables

Listings

Bibliography

[AAC09] Jason Ansel, Kapil Arya, and Gene Cooperman. DMTCP: transparent checkpointing for cluster computations and the desktop. In *23rd IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, pages 1–12, 2009.

[AC01] Giuseppe Attardi and Antonio Cisternino. Reflection support by means of template metaprogramming. In *Proceedings of Third International Conference on Generative and Component-Based Software Engineering, LNCS*, pages 118–127, 2001.

[ACE$^+$12] Mehdi Amini, Béatrice Creusillet, Stéphanie Even, Ronan Keryell, Onig Goubier, Serge Guelton, Janice Onanian McMahon, François-Xavier Pasquier, Grégoire Péan, and Pierre Villalon. Par4all: From convex array regions to heterogeneous computing. In *IMPACT 2012: Second International Workshop on Polyhedral Compilation Techniques HiPEAC*, 2012.

[ADY12] Z. Abdelhafidi, M. Djoudi, and M. B. Yagoubi. An improved schema of coordinated checkpointing protocol for distributed systems based on popular process. In *2012 International Conference on Innovations in Information Technology (IIT)*, pages 367–372, March 2012.

[AWE$^+$09] Hasan Abbasi, Matthew Wolf, Greg Eisenhauer, Scott Klasky, Karsten Schwan, and Fang Zheng. Datastager: Scalable data staging services for petascale applications. In *Proceedings of the 18th ACM International Symposium on High Performance Distributed Computing*, HPDC '09, pages 39–48, New York, NY, USA, 2009. ACM.

[BBC$^+$08] Keren Bergman, Shekhar Borkar, Dan Campbell, William Carlson, William Dally, Monty Denneau, Paul Franzon, William Harrod, Jon Hiller, Sherman Karp, Stephen Keckler, Dean Klein, Robert Lucas, Mark Richards, Al Scarpelli, Steven Scott, Allan Snavely, Thomas Sterling, R. Stanley Williams, Katherine Yelick, Keren Bergman, Shekhar Borkar, Dan Campbell, William Carlson, William Dally, Monty Denneau, Paul Franzon, William Harrod, Jon Hiller, Stephen Keckler, Dean Klein, Peter Kogge, R. Stanley Williams, and Katherine Yelick. Exascale computing study: Technology challenges in achieving exascale systems peter kogge, editor & study lead, 2008.

[BBK$^+$12] Georg Birkenheuer, André Brinkmann, Jürgen Kaiser, Axel Keller, Matthias Keller, Christoph Kleineweber, Christoph Konersmann, Oliver Niehörster, Thorsten Schäfer, Jens Simon, et al. Virtualized hpc: a contradiction in terms? *Software: Practice and Experience*, 42(4):485–500, 2012.

[BE09] A. Brinkmann and D. Eschweiler. A microdriver architecture for error correcting codes inside the linux kernel. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 35:1–35:10, New York, NY, USA, 2009. ACM.

[Bel05] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *FREENIX Track: 2005 USENIX Annual Technical Conference*, pages 41–46, 2005.

[BELL09] Deepavali Bhagwat, Kave Eshghi, Darrell D. E. Long, and Mark Lillibridge. Extreme Binning: Scalable, Parallel Deduplication for Chunk-based File Backup. In *Proceedings of the 17th IEEE International Symposium on Modeling, Analysis, and Simulation (MASCOTS)*, pages 1–9, 2009.

[BGG$^+$09] John Bent, Garth Gibson, Gary Grider, Ben McClelland, Paul Nowoczynski, James Nunez, Milo Polte, and Meghan Wingate. Plfs: A checkpoint filesystem for parallel applications. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 21:1–21:12, New York, NY, USA, 2009. ACM.

[BGTK$^+$11] Leonardo Bautista-Gomez, Seiji Tsuboi, Dimitri Komatitsch, Franck Cappello, Naoya Maruyama, and Satoshi Matsuoka. Fti: High performance fault tolerance interface for hybrid systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 32:1–32:32, New York, NY, USA, 2011. ACM.

[BHMR97] R. Baldoni, J. M. Helary, A. Mostefaoui, and M. Raynal. A communication-induced checkpointing protocol that ensures rollback-dependency trackability. In *Proceedings of IEEE 27th International Symposium on Fault Tolerant Computing*, pages 68–77, June 1997.

[BHRS08] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 101–113, New York, NY, USA, 2008. ACM.

[BLC10] Sébastien Boisvert, François Laviolette, and Jacques Corbeil. Ray: Simultaneous Assembly of Reads from a Mix of High-Throughput Sequencing Technologies. *Journal of Computational Biology*, 17(11):1519–1533, 2010.

[BMP$^+$04] Greg Bronevetsky, Daniel Marques, Keshav Pingali, Peter K. Szwed, and Martin Schulz. Application-level checkpointing for shared memory programs. In *11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 235–247, 2004.

[BMPS03a] Greg Bronevetsky, Daniel Marques, Keshav Pingali, and Paul Stodghill. Automated application-level checkpointing of mpi programs. In *Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '03, pages 84–94, New York, NY, USA, 2003. ACM.

[BMPS03b] Greg Bronevetsky, Daniel Marques, Keshav Pingali, and Paul Stodghill. Collective operations in an application-level fault tolerant mpi system. In *International Conference on Supercomputing (ICS)*, pages 234–243, 2003.

[BPS06] Greg Bronevetsky, Keshav Pingali, and Paul Stodghill. Experimental evaluation of application-level checkpointing for openmp programs. In *Proceedings of the 20th Annual International Conference on Supercomputing*, ICS '06, pages 2–13, New York, NY, USA, 2006. ACM.

[BPWM16] Jens Breitbart, Simon Pickartz, Josef Weidendorfer, and Antonello Monti. Viability of Virtual Machines in HPC. In *Euro-Par 2016: Parallel Processing Workshops*, Lecture Notes in Computer Science. Springer International Publishing, 2016. Accepted for puplication.

[BWPB11] T. Beisel, T. Wiersema, C. Plessl, and A. Brinkmann. Cooperative multitasking for heterogeneous accelerators in the linux completely fair scheduler. In *ASAP 2011 - 22nd IEEE International Conference on Application-specific Systems, Architectures and Processors*, pages 223–226, Sept 2011.

[BWSS12] Sean Barker, Timothy Wood, Prashant Shenoy, and Ramesh Sitaraman. An Empirical Study of Memory Sharing in Virtual Machines. In *Proceedings of the 2012 USENIX Annual Technical Conference (ATC 12)*, pages 273–284, 2012.

[BWT15] Jens Breitbart, Josef Weidendorfer, and Carsten Trinitis. Case study on co-scheduling for HPC applications. In *44th International Conference on Parallel Processing Workshops (ICPPW)*, pages 277–285, 2015.

[Cer03] Paul E Ceruzzi. *A history of modern computing.* MIT press, 2003.

[CFH$^+$05] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *Proceedings of the 2Nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2*, NSDI'05, pages 273–286, 2005.

[CHL$^+$06] Camille Coti, Thomas Herault, Pierre Lemarinier, Laurence Pilard, Ala Rezmerita, Eric Rodriguez, and Franck Cappello. Blocking vs. non-blocking coordinated checkpointing for large-scale fault tolerant mpi. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM.

[CL85] K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, February 1985.

[Cla07] S Clara. Nvidia, cuda (compute unified device architecture) programming guide, 2007.

[CLG$^+$94] Peter M. Chen, Edward K. Lee, Garth A. Gibson, Randy H. Katz, and David A. Patterson. Raid: High-performance, reliable secondary storage. *ACM Comput. Surv.*, 26(2):145–185, June 1994.

[CPG13] Chuck Cranor, Milo Polte, and Garth Gibson. Structuring plfs for extensibility. In *Proceedings of the 8th Parallel Data Storage Workshop*, PDSW '13, pages 20–26, New York, NY, USA, 2013. ACM.

[CRMG12] I. Cores, G. Rodriguez, M.J. Martin, and P. Gonz'lez. Reducing application-level checkpoint file sizes: Towards scalable fault tolerance solutions. In *IEEE 10th International Symposium on Parallel and Distributed Processing with Applications (ISPA)*, pages 371–378, 2012.

[CS98] Guohong Cao and M. Singhal. On coordinated checkpointing in distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, 9(12):1213–1225, Dec 1998.

[CSWB08] Matthew L Curry, Anthony Skjellum, H Lee Ward, and Ron Brightwell. Accelerating reed-solomon coding in raid systems with gpus. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–6. IEEE, 2008.

[CV13] Selim Ciraci and Oreste Villa. Exploiting points-to maps for de-/serialization code generation. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, SAC '13, pages 1712–1719, New York, NY, USA, 2013. ACM.

[DA06] Dinakar Dhurjati and Vikram Adve. Backwards-compatible array bounds checking for C with very low overhead. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 162–171, New York, NY, USA, 2006. ACM.

[DBB07] Romain Dolbeau, Stéphane Bihan, and François Bodin. Hmpp: A hybrid multi-core parallel programming environment. In *Workshop on general purpose processing on graphics processing units (GPGPU 2007)*, volume 28. Citeseer, 2007.

[DBM$^+$11] Jack Dongarra, Pete Beckman, Terry Moore, Patrick Aerts, Giovanni Aloisio, Jean-Claude Andre, David Barkai, Jean-Yves Berthou, Taisuke Boku, Bertrand Braunschweig, Franck Cappello, Barbara Chapman, Xuebin Chi, Alok Choudhary, Sudip Dosanjh, Thom Dunning, Sandro Fiore, Al Geist, Bill Gropp, Robert Harrison, Mark Hereld, Michael Heroux, Adolfy Hoisie, Koh Hotta, Zhong Jin, Yutaka Ishikawa, Fred Johnson, Sanjay Kale, Richard Kenway, David Keyes, Bill Kramer, Jesus Labarta, Alain Lichnewsky, Thomas Lippert, Bob Lucas, Barney Maccabe, Satoshi Matsuoka, Paul Messina, Peter Michielse, Bernd Mohr, Matthias S. Mueller, Wolfgang E. Nagel, Hiroshi Nakashima, Michael E Papka, Dan Reed, Mitsuhisa Sato, Ed Seidel, John Shalf, David Skinner, Marc Snir, Thomas Sterling, Rick Stevens, Fred Streitz, Bob Sugar, Shinji Sumimoto, William Tang, John Taylor, Rajeev Thakur, Anne Trefethen, Mateo Valero, Aad Van Der Steen, Jeffrey Vetter, Peg Williams, Robert Wisniewski, and Kathy Yelick. The international exascale software project roadmap. *Int. J. High Perform. Comput. Appl.*, 25(1):3–60, February 2011.

[DCF03] Aaron Darling, Lucas Carey, and Wu-Chun Feng. The design, implementation, and evaluation of mpiBLAST. *ClusterWorld Conference & Expo and the 4th International Conference on Linux Cluster: The HPC Revolution 2003*, pages 13–15, June 2003.

[DKA06] Dinakar Dhurjati, Sumant Kowshik, and Vikram Adve. Safecode: enforcing alias analysis for weakly typed languages. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 144–157, New York, NY, USA, 2006. ACM.

[DM98] Leonardo Dagum and Ramesh Menon. Openmp: An industry-standard api for shared-memory programming. *IEEE Comput. Sci. Eng.*, 5(1):46–55, January 1998.

[DS11] Julien Dusser and André Seznec. Decoupled zero-compressed memory. In *6th International Conference on High Performance Embedded Architectures and Compilers (HiPEAC)*, pages 77–86, 2011.

[Due03] Jason Duell. The design and implementation of berkeley lab's linux checkpoint/restart. Technical report, Lawrence Berkeley National Laboratory, 2003.

[EAWJ02] Elmootazbellah Nabil Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys (CSUR)*, 34(3):375–408, 2002.

[EP04] Elmootazbellah N Elnozahy and James S Plank. Checkpointing for peta-scale systems: A look into the future of practical rollback-recovery. *IEEE Transactions on Dependable and Secure Computing*, 1(2):97–108, 2004.

[ES05] Magnus Ekman and Per Stenstrom. A robust main-memory compression scheme. In *Proceedings of the 32Nd Annual International Symposium on Computer Architecture*, ISCA '05, pages 74–85, Washington, DC, USA, 2005. IEEE Computer Society.

[Far16] R. Farber. *Parallel Programming with OpenACC*. Elsevier Science, 2016.

[Fer07] Justin N. Ferguson. Understanding the heap by breaking it. *Black Hat USA*, pages 1–39, 2007.

[Fer10] Leo Ferres. Memory management in c: The heap and the stack. *Department of Computer Science, Universidad de Concepcion*, 2010.

[FFRR15] Wes Felter, Alexandre Ferreira, Ram Rajamony, and Juan Rubio. An updated performance comparison of virtual machines and linux containers. In *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium on*, pages 171–172. IEEE, 2015.

[FL05] A Murat Fiskiran and Ruby B Lee. Fast parallel table lookups to accelerate symmetric-key cryptography. In *Information Technology: Coding and Computing, 2005. ITCC 2005. International Conference on*, volume 1, pages 526–531. IEEE, 2005.

[Fro13] Scott Froberg. Distributed and cloud computing from parallel processing to the internet of things by kai hwang, geoffry c. fox, and jack j. dongarra. *SIGSOFT Softw. Eng. Notes*, 38(2):34–34, March 2013.

[FWL$^+$14] K. B. Ferreira, P. Widener, S. Levy, D. Arnold, and T. Hoefler. Understanding the effects of communication and coordination on checkpointing at scale. In *SC14: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 883–894, Nov 2014.

[FYP93] David W. Flater, Yelena Yesha, and E. K. Park. Extensions to the C programming language for enhanced fault detection. *Softw. Pract. Exper.*, 23(6):617–628, June 1993.

[GBB$^+$09] Paolo Giannozzi, Stefano Baroni, Nicola Bonini, Matteo Calandra, Roberto Car, Carlo Cavazzoni, Davide Ceresoli, Guido L Chiarotti, Matteo Cococcioni, Ismaila Dabo, Andrea Dal Corso, Stefano de Gironcoli, Stefano Fabris, Guido Fratesi, Ralph Gebauer, Uwe Gerstmann, Christos Gougoussis, Anton Kokalj, Michele Lazzeri, Layla Martin-Samos, Nicola Marzari, Francesco Mauri, Riccardo Mazzarello, Stefano Paolini, Alfredo Pasquarello, Lorenzo Paulatto, Carlo Sbraccia, Sandro Scandolo, Gabriele Sclauzero, Ari P Seitsonen, Alexander Smogunov, Paolo Umari, and Renata M Wentzcovitch. Quantum espresso: a modular and open-source software project for quantum simulations of materials. *Journal of Physics: Condensed Matter*, 21(39), 2009.

[GC11] A. Guermouche and F. Cappello. Uncoordinated checkpointing without domino effect for send-deterministic mpi applications. In *2011 IEEE International Parallel Distributed Processing Symposium*, pages 989–1000, May 2011.

[GCR+07] J. N. Glosli, K. J. Caspersen, R. E. Rudd, J. A. Gunnels, and F. H. Streitz. 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, pages 58:1–58:11, New York, NY, USA, 2007. ACM.

[Gib92] Garth Alan Gibson. Redundant disk arrays: Reliable, parallel secondary storage. 1992.

[GL93] W. Gropp and E. Lusk. The mpi communication library: its design and a portable implementation. In *Proceedings of Scalable Parallel Libraries Conference*, pages 160–165, Oct 1993.

[GPS+16] Ramy Gad, Simon Pickartz, Tim Süß, Lars Nagel, Stefan Lankes, and André Brinkmann. Accelerating application migration in hpc. In *International Conference on High Performance Computing*, pages 663–673. Springer, 2016.

[GSB14] Ramy Gad, Tim Süß, and André Brinkmann. Compiler driven automatic kernel context migration for heterogeneous computing. In *Distributed Computing Systems (ICDCS), 2014 IEEE 34th International Conference on*, pages 389–398, June 2014.

[GSjP05] R. Gioiosa, J.C. Sancho, s. jiang, and F. Petrini. Transparent, incremental checkpointing at kernel level: a foundation for fault tolerance for parallel computers. In *Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 Conference*, pages 9–9, November 2005.

[HA09] Tianyi David Han and Tarek S. Abdelrahman. hicuda: A high-level directive-based language for gpu programming. In *Proceedings of 2Nd Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-2, pages 52–61, New York, NY, USA, 2009. ACM.

[HBL+13] Jonathan D. Halverson, Thomas Brandes, Olaf Lenz, Axel Arnold, Staš Bevc, Vitaliy Starchenko, Kurt Kremer, Torsten Stuehn, and Dirk Reith. ESPResSo++: A modern multiscale simulation package for soft matter systems. *Computer Physics Communications*, 184(4):1129–1149, 2013.

[HGLP07] Wei Huang, Qi Gao, Jiuxing Liu, and Dhabaleswar K. Panda. High performance virtual machine migration with RDMA over modern interconnects. In *IEEE International Conference on Cluster Computing*, pages 11–20, 2007.

[HGSZ10] Jinhua Hu, Jianhua Gu, Guofei Sun, and Tianhai Zhao. A scheduling strategy on load balancing of virtual machine resources in cloud computing environment. In *3rd International Symposium on Parallel Architectures, Algorithms and Programming (PAAP)*, pages 89–96, 2010.

[HISV14] Jürg Hutter, Marcella Iannuzzi, Florian Schiffmann, and Joost VandeVondele. cp2k: atomistic simulations of condensed matter systems. *Wiley Interdisciplinary Reviews: Computational Molecular Science*, 4(1):15–25, 2014.

[HJ91] Reed Hastings and Bob Joyce. Purify: Fast detection of memory leaks and access errors. In *Proc. of the Winter 1992 USENIX Conference*, pages 125–138, 1991.

[HKVDSL08] Berk Hess, Carsten Kutzner, David Van Der Spoel, and Erik Lindahl. Gromacs 4: algorithms for highly efficient, load-balanced, and scalable molecular simulation. *Journal of chemical theory and computation*, 4(3):435–447, 2008.

[HNIS11] Takahiro Hirofuchi, Hidemoto Nakada, Satoshi Itoh, and Satoshi Sekiguchi. Reactive consolidation of virtual machines enabled by postcopy live migration. In *5th International Workshop on Virtualization Technologies in Distributed Computing, VTDC@HPDC 2011*, pages 11–18, 2011.

[IAB$^+$12] D. Ibtesham, D. Arnold, P. G. Bridges, K. B. Ferreira, and R. Brightwell. On the viability of compression for reducing the overheads of checkpoint/restart-based fault tolerance. In *2012 41st International Conference on Parallel Processing*, pages 148–157, September 2012.

[IAFB12] Dewan Ibtesham, Dorian Arnold, Kurt B. Ferreira, and Patrick G. Bridges. 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, pages 302–311, Berlin, Heidelberg, 2012. Springer-Verlag.

[IMB$^+$12] Tanzima Zerin Islam, Kathryn Mohror, Saurabh Bagchi, Adam Moody, Bronis R. de Supinski, and Rudolf Eigenmann. 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, pages 17:1–17:11. IEEE, 2012.

[JHM16] Richard Jones, Antony Hosking, and Eliot Moss. *The garbage collection handbook: the art of automatic memory management.* CRC Press, 2016.

[JI$^+$12] Amue Gonewa John, Umoh Godwin Ikpe, et al. Dynamic behavior in customersâ$^{\text{TM}}$ switching and market share analysis: The markov model perspectives. *Global Journal of Management And Business Research*, 12(17), 2012.

[JJT07] Hrvoje Jasak, Aleksandar Jemcov, and Zeljko Tukovic. OpenFOAM: A C++ library for complex physics simulations. In *Proceedings of the International Workshop on Coupled Methods in Numerical Dynamics*, August 2007.

[JK97] Richard W M Jones and Paul H J Kelly. Backwards-compatible bounds checking for arrays and pointers in c programs. In *Distributed Enterprise Applications. HP Labs Tech Report*, pages 255–283, 1997.

[JKCS12] Hui Jin, Tao Ke, Yong Chen, and Xian-He Sun. 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, pages 276–283, Washington, DC, USA, 2012. IEEE Computer Society.

[KB02] F. Karablieh and R.A. Bazzi. Heterogeneous checkpointing for multithreaded applications. In *Reliable Distributed Systems, 2002. Proceedings. 21st IEEE Symposium on*, pages 140–149, 2002.

[KDK$^+$11] S. W. Keckler, W. J. Dally, B. Khailany, M. Garland, and D. Glasco. Gpus and the future of parallel computing. *IEEE Micro*, 31(5):7–17, September 2011.

[KF10] G. Kaur and M. M. Fuad. An evaluation of protocol buffer. In *Proceedings of the IEEE SoutheastCon 2010 (SoutheastCon)*, pages 459–462, March 2010.

[KGS$^+$16] Jürgen Kaiser, Ramy Gad, Tim Süß, Federico Padua, Lars Nagel, and André Brinkmann. Deduplication potential of hpc applications' checkpoints. In *Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER) 2016, Taipei, Taiwan, Sept. 2016*, Septemper 2016.

[Kim05] Byoung-Jip Kim. Comparison of the existing checkpoint systems. Technical report, IBM Watson, 2005.

[KK93] Laxmikant V. Kale and Sanjeev Krishnan. Charm++: A portable concurrent object oriented system based on c++. In *Proceedings of the Eighth Annual Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '93, pages 91–108. ACM, 1993.

[KKL$^+$07] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm: the Linux Virtual Machine Monitor. In *Linux Symposium*, pages 225–230, June 2007.

[KMBS15] Jürgen Kaiser, Dirk Meister, André Brinkmann, and Tim Süß. Deriving and Comparing Deduplication Techniques Using a Model-Based Classification. In *Proceedings of the European Conference on Computer Systems (EuroSys)*. ACM, 2015.

[KMI$^+$12] Abhishek Kulkarni, Adam Manzanares, Latchesar Ionkov, Michael Lang, and Andrew Lumsdaine. The design and implementation of a multi-level content-addressable checkpoint file system. In *Proceedings of the 19th International Conference on High Performance Computing (HiPC)*, pages 1–10. IEEE, 2012.

[KP93] J. L. Kim and T. Park. An efficient protocol for checkpointing recovery in distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, 4(8):955–960, Aug 1993.

[KS97] G. P. Kavanaugh and W. H. Sanders. Performance analysis of two time-based coordinated checkpointing protocols. In *Proceedings Pacific Rim International Symposium on Fault-Tolerant Systems*, pages 194–201, Dec 1997.

[KS02] Michael Kozuch and Mahadev Satyanarayanan. Internet suspend/resume. In *4th IEEE Workshop on Mobile Computing Systems and Applications (WMCSA)*, 2002.

[KT87] R. Koo and S. Toueg. Checkpointing and rollback-recovery for distributed systems. *IEEE Transactions on Software Engineering*, SE-13(1):23–31, Jan 1987.

[Kur01] T. Kuroda. Cmos design challenges to power wall. In *Digest of Papers. Microprocesses and Nanotechnology 2001. 2001 International Microprocesses and Nanotechnology Conference (IEEE Cat. No.01EX468)*, pages 6–7, Oct 2001.

[LA04] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.

[LD09] Heng Li and Richard Durbin. Fast and accurate short read alignment with Burrows–Wheeler transform. *Bioinformatics*, 25(14):1754–1760, 2009.

[LEB$^+$09] Mark Lillibridge, Kave Eshghi, Deepavali Bhagwat, Vinay Deolalikar, Greg Trezis, and Peter Camble. Sparse Indexing: Large Scale, Inline Deduplication Using Sampling and Locality. In *Fast*, volume 9, pages 111–123, 2009.

[Lee03] Hanho Lee. High-speed vlsi architecture for parallel reed-solomon decoder. *IEEE transactions on very large scale integration (VLSI) systems*, 11(2):288–294, 2003.

[LkLC$^+$03] Jianwei Li, Wei keng Liao, Alok Choudhary, Robert Ross, Rajeev Thakur, William Gropp, Rob Latham, Andrew Siegel, Brad Gallagher, and Michael Zingale. Parallel netCDF: A High-Performance Scientific I/O Interface. *SC Conference*, 0:39, 2003.

[LLB09] Nicolas Lartillot, Thomas Lepage, and Samuel Blanquart. PhyloBayes 3: a Bayesian software package for phylogenetic reconstruction and molecular dating. *Bioinformatics*, 25(17):2286–2288, 2009.

[Llva] clang: a C language family frontend for LLVM. `http://clang.llvm.org/index.html`.

[Llvb] LLVM Language Reference Manual. `http://llvm.org/docs/LangRef.html`.

[Llvc] LLVM's Analysis and Transform Passes. `http://llvm.org/docs/Passes.html`.

[LNP94] K. Li, J. F. Naughton, and J. S. Plank. Low-latency, concurrent checkpointing for parallel programs. *IEEE Trans. Parallel Distrib. Syst.*, 5(8):874–879, August 1994.

[LTPS09] Ben Langmead, Cole Trapnell, Mihai Pop, and Steven Salzberg. Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome Biology*, 10(3), 2009.

[LY87] Ten H Lai and Tao H Yang. On distributed snapshots. *Information Processing Letters*, 25(3):153–158, 1987.

[LZS$^+$06] Yinglung Liang, Yanyong Zhang, A. Sivasubramaniam, M. Jette, and R. Sahoo. Bluegene/l failure analysis and prediction models. In *International Conference on Dependable Systems and Networks (DSN'06)*, pages 425–434, June 2006.

[MB09] Dirk Meister and André Brinkmann. Multi-level comparison of data deduplication in a backup scenario. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, SYSTOR '09, pages 8:1–8:12, New York, NY, USA, 2009. ACM.

[Mbg] Microbial Genome Database for Comparative Analaysis. `http://mbgd.genome.ad.jp/`.

[MBMS10] Adam Moody, Greg Bronevetsky, Kathryn Mohror, and Bronis R. de Supinski. Design, modeling, and evaluation of a scalable multi-level checkpointing system. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[MCM01] Athicha Muthitacharoen, Benjie Chen, and David Mazières. A low-bandwidth network file system. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, SOSP '01, pages 174–187, 2001.

[MDP$^+$00] Dejan S. Milojičić, Fred Douglis, Yves Paindaveine, Richard Wheeler, and Songnian Zhou. Process migration. *ACM Comput. Surv.*, 32(3):241–299, September 2000.

[Meh08] Vijay P Mehta. Getting started with object-relational mapping. *Pro LINQ Object Relational Mapping with C# 2008*, pages 3–15, 2008.

[Mei11] Dirk Meister. FS-C, File System Chunking Tool Suite, version 0.3.9. `https://github.com/dmeister/fs-c`, 2011.

[Mer14] Dirk Merkel. Docker: Lightweight linux containers for consistent development and deployment. *Linux J.*, 2014(239), March 2014.

[MFR$^+$13] Konrad Miller, Fabian Franz, Marc Rittinghaus, Marius Hillenbrand, and Frank Bellosa. Xlh: More effective memory deduplication scanners through cross-layer hints. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 279–290. USENIX, 2013.

[MG09] Andrew Maloney and Andrzej Goscinski. A survey and review of the current state of rollback-recovery for cluster systems. *Concurrency and Computation: Practice and Experience*, 21(12):1632–1666, 2009.

[MKB$^+$12] Dirk Meister, Jürgen Kaiser, André Brinkmann, Toni Cortes, Michael Kuhn, and Julian Kunkel. A Study on Data Deduplication in HPC Storage Systems. In *Proceedings of International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 7:1–7:11. IEEE, 2012.

[MKB13] Dirk Meister, Jürgen Kaiser, and André Brinkmann. Block locality caching for data deduplication. In *Proceedings of the 6th International Systems and Storage Conference*, page 15. ACM, 2013.

[MKK15] Roberto Morabito, Jimmy Kjällman, and Miika Komu. Hypervisors vs. lightweight virtualization: A performance comparison. In *Proceedings of the 2015 IEEE International Conference on Cloud Engineering*, IC2E '15, pages 386–393, Washington, DC, USA, 2015. IEEE Computer Society.

[MMBdS14] K. Mohror, A. Moody, G. Bronevetsky, and B. R. de Supinski. Detailed modeling and evaluation of a scalable multilevel checkpointing system. *IEEE Transactions on Parallel and Distributed Systems*, 25(9):2255–2263, Sept 2014.

[MMdS12] K. Mohror, A. Moody, and B. R. de Supinski. Asynchronous checkpoint migration with mrnet in the scalable checkpoint / restart library. In *IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN 2012)*, pages 1–6, June 2012.

[MNB$^+$15] Markus Mäsker, Lars Nagel, André Brinkmann, Foad Lotfifar, and Matthew Johnson. Smart grid-aware scheduling in data centres. In *2015 Sustainable Internet and ICT for Sustainability (SustainIT)*, pages 1–9, 2015.

[Moo00] Gordon E. Moore. Readings in computer architecture. chapter Cramming More Components Onto Integrated Circuits, pages 56–59. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000.

[MSFS09] John Mehnert-Spahn, Eugen Feller, and Michael Schoettner. Incremental checkpointing for grids. In *Linux Symposium*, volume 120. Citeseer, 2009.

[MV15] Sparsh Mittal and Jeffrey S. Vetter. A survey of cpu-gpu heterogeneous computing techniques. *ACM Comput. Surv.*, 47(4):69:1–69:35, July 2015.

[NB10] Gene Novark and Emery D. Berger. Dieharder: securing the heap. In *Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS 2010, Chicago, Illinois, USA, October 4-8, 2010*, pages 573–584, 2010.

[NC13] Bogdan Nicolae and Franck Cappello. Ai-ckpt: Leveraging memory access patterns for adaptive asynchronous incremental checkpointing. In *Proceedings of the 22Nd International Symposium on High-performance Parallel and Distributed Computing*, HPDC '13, pages 155–166, New York, NY, USA, 2013. ACM.

[Nic13] Bogdan Nicolae. Towards scalable checkpoint restart: A collective inline memory contents deduplication proposal. In *Proceedings of the 27th International Symposium on Parallel and Distributed Processing*, IPDPS '13, pages 19–28. IEEE Computer Society, 2013.

[NMES07] Arun Babu Nagarajan, Frank Mueller, Christian Engelmann, and Stephen L. Scott. Proactive fault tolerance for HPC with xen virtualization. In *21st Annual International Conference on Supercomputing (ICS)*, pages 23–32, 2007.

[NWK05] S. Naffziger, J. Warnock, and H. Knapp. Se2 when processors hit the power wall (or "when the cpu hits the fan"). In *ISSCC. 2005 IEEE International Digest of Technical Papers. Solid-State Circuits Conference, 2005.*, pages 16–17, Feb 2005.

[OAFI04] M. Ohara, M. Arai, S. Fukumoto, and K. Iwasaki. Finding a recovery line in uncoordinated checkpointing. In *24th International Conference on Distributed Computing Systems Workshops, 2004. Proceedings.*, pages 628–633, March 2004.

[ORS$^+$05] Adam J Oliner, Larry Rudolph, Ramendra K Sahoo, José E Moreira, and Manish Gupta. Probabilistic qos guarantees for supercomputing systems. In *Dependable Systems and Networks, 2005. DSN 2005. Proceedings. International Conference on*, pages 634–643. IEEE, 2005.

[ORS06a] Adam Oliner, Larry Rudolph, and Ramendra Sahoo. Cooperative checkpointing theory. In *Proceedings of the 20th International*

Conference on Parallel and Distributed Processing, IPDPS'06, pages 132–132, Washington, DC, USA, 2006. IEEE Computer Society.

[ORS06b] Adam J. Oliner, Larry Rudolph, and Ramendra K. Sahoo. Cooperative checkpointing: A robust approach to large-scale systems reliability. In *Proceedings of the 20th Annual International Conference on Supercomputing*, ICS '06, pages 14–23, New York, NY, USA, 2006. ACM.

[Pat11] David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[PBKL95] James S. Plank, Micah Beck, Gerry Kingsley, and Kai Li. Libckpt: Transparent checkpointing under UNIX. In *USENIX 1995 Technical Conference on UNIX and Advanced Computing Systems*, pages 213–224, 1995.

[PBW+05] James C. Phillips, Rosemary Braun, Wei Wang, James Gumbart, Emad Tajkhorshid, Elizabeth Villa, Christophe Chipot, Robert D. Skeel, Laxmikant Kalé, and Klaus Schulten. Scalable molecular dynamics with NAMD. *Journal of Computational Chemistry*, 26(16):1781–1802, 2005.

[PCL+99] James S. Plank, Yuqun Chen, Kai Li, Micah Beck, and Gerry Kingsley. Memory exclusion: Optimizing the performance of checkpointing systems. *Softw. Pract. Exper.*, 29(2):125–142, February 1999.

[PCL+16] Simon Pickartz, Carsten Clauss, Stefan Lankes, Stephan Krempel, Thomas Moschny, and Antonello Monti. Non-intrusive migration of MPI processes in os-bypass networks. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPS Workshops 2016, Chicago, IL, USA, May 23-27, 2016*, pages 1728–1735, 2016.

[PE98] J. S. Plank and W. R. Elwasif. Experimental assessment of workstation failures and their impact on checkpointing systems. In *28th International Symposium on Fault-Tolerant Computing*, pages 48–57, Munich, June 1998.

[PGK88] David A. Patterson, Garth Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (raid). In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, SIGMOD '88, pages 109–116, New York, NY, USA, 1988. ACM.

[PGL+14] Simon Pickartz, Ramy Gad, Stefan Lankes, Lars Nagel, Tim Süß, André Brinkmann, and Stephan Krempel. *Migration Techniques in HPC Environments*, pages 486–497. Springer International Publishing, Cham, 2014.

[PL94] J. S. Plank and Kai Li. ickp: a consistent checkpointer for multicomputers. *IEEE Parallel Distributed Technology: Systems Applications*, 2(2):62–67, Summer 1994.

[Pla97] James S. Plank. A tutorial on reed-solomon coding for fault-tolerance in raid-like systems. *Softw. Pract. Exper.*, 27(9):995–1012, September 1997.

[Pli95] Steve Plimpton. Fast parallel algorithms for short-range molecular dynamics. *Journal of Computational Physics*, 117(1):1–19, 1995.

[PLP98] James S. Plank, Kai Li, and Michael A. Puening. Diskless checkpointing. *IEEE Trans. Parallel Distrib. Syst.*, 9(10):972–986, October 1998.

[PLQL12] Darren Peters, Xuemei Luo, Ke Qiu, and Ping Liang. Speeding up large-scale next generation sequencing data analysis with pbwa. *Journal of Applied Bioinformatics & Computational Biology*, 1(1), 2012.

[Pma] pMap: Parallel Sequence Mapping Tool. `http://bmi.osu.edu/hpc/software/pmap/pmap.html`.

[Pou17] Behnam Pourghassemi. *cudaCR: An In-kernel Application-level Checkpoint/Restart Scheme for CUDA Applications.* PhD thesis, University of California, Irvine, 2017.

[PSW08] Joseph M. Prusa, Piotr K. Smolarkiewicz, and Andrzej A. Wyszogrodzki. Eulag, a computational model for multiscale flows. *Computers & Fluids*, 37(9):1193–1207, 2008.

[PT01] James S. Plank and Michael G. Thomason. Processor allocation and checkpoint interval selection in cluster computing systems. *J. Parallel Distrib. Comput.*, 61(11):1570–1590, November 2001.

[QLI+17] Yingjin Qian, Xi Li, Shuichi Ihara, Lingfang Zeng, Jürgen Kaiser, Tim Süß, and André Brinkmann. A configurable rule based classful token bucket filter network request scheduler for the lustre file system. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '17, pages 6:1–6:12, New York, NY, USA, 2017. ACM.

[Rab81] Michael O. Rabin. Fingerprinting by Random Polynomials. Technical report, Center for Research in Computing Technology, Harvard University, 1981.

[RAM03] P. C. Roth, D. C. Arnold, and B. P. Miller. Mrnet: A software-based multicast/reduction network for scalable tools. In *Supercomputing, 2003 ACM/IEEE Conference*, pages 21–21, Nov 2003.

[RD10] Nathan Regola and Jean-Christophe Ducom. Recommendations for virtualization technologies in high performance computing. In *Cloud Computing Technology and Science (CloudCom), 2010*

IEEE Second International Conference on, pages 409–416. IEEE, 2010.

[RD14] Daniel A. Reed and Jack Dongarra. Exascale computing and big data: The next frontier. In *submitted Communications of the ACM*, April 2014.

[RGU+11] Thomas Ropars, Amina Guermouche, Bora Uçar, Esteban Meneses, Laxmikant V. Kalé, and Franck Cappello. On the use of cluster-based partial message logging to improve fault tolerance for mpi hpc applications. In *Proceedings of the 17th International Conference on Parallel Processing - Volume Part I*, Euro-Par'11, pages 567–578, Berlin, Heidelberg, 2011. Springer-Verlag.

[RH11] F. Romero and T. J. Hacker. Live migration of parallel applications with openvz. In *2011 IEEE Workshops of International Conference on Advanced Information Networking and Applications*, pages 526–531, March 2011.

[RLT10] Martin Randles, David J. Lamb, and A. Taleb-Bendiab. A comparative study into distributed load balancing algorithms for cloud computing. In *24th IEEE International Conference on Advanced Information Networking and Applications Workshops (WAINA)*, pages 551–556, 2010.

[RMMP13] Raghunath Rajachandrasekar, Adam Moody, Kathryn Mohror, and Dhabaleswar K. (DK) Panda. A 1 pb/s file system to checkpoint three million mpi tasks. In *Proceedings of the 22Nd International Symposium on High-performance Parallel and Distributed Computing*, HPDC '13, pages 143–154, New York, NY, USA, 2013. ACM.

[Rom02] Eric Roman. A survey of checkpoint/restart implementations. Technical report, Lawrence Berkeley National Laboratory, Tech, 2002.

[SBF+04] Martin Schulz, Greg Bronevetsky, Rohit Fernandes, Daniel Marques, Keshav Pingali, and Paul Stodghill. Implementation and evaluation of a scalable application-level checkpoint-recovery scheme for MPI programs. In *ACM/IEEE SC Conference on High Performance Networking and Computing*, page 38, 2004.

[SDG+16a] Tim Süß, Nils Döring, Ramy Gad, Lars Nagel, André Brinkmann, Dustin Feld, Eric Schröder, and Thomas Soddemann. Impact of the scheduling strategy in heterogeneous systems that provide co-scheduling. In *1st COSH Workshop on Co-Scheduling of HPC Applications, COSH@HiPEAC 2016*, pages 37–42, 2016.

[SDG+16b] Tim Süß, Nils Döring, Ramy Gad, Lars Nagel, André Brinkmann, Dustin Feld, Thomas Soddemann, and Stefan Lankes. Varysched: A framework for variable scheduling in heterogeneous environments. In *2016 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 489–492, September 2016.

[SDG$^+$17] Tim Süß, Nils Döring, Ramy Gad, Lars Nagel, André Brinkmann, Dustin Feld, Eric Schricker, and Thomas Soddemann. *Impact of the scheduling strategy in heterogeneous systems that provide co-scheduling*, pages 142–162. IOS Press, 2017.

[SG$^+$07a] Mahadev Satyanarayanan, Benjamin Gilbert, et al. Pervasive personal computing in an internet suspend/resume system. *IEEE Internet Computing*, 11(2):16–25, 2007.

[SG07b] B. Schroeder and G. A. Gibson. Understanding failures in petascale computers. In *Journal of Physics Conference Series*, volume 78 of *Journal of Physics Conference Series*, page 012022, July 2007.

[SGE$^+$13] Bjorn Stevens, Marco Giorgetta, Monika Esch, Thorsten Mauritsen, Traute Crueger, Sebastian Rast, Marc Salzmann, Hauke Schmidt, Jürgen Bader, Karoline Block, et al. Atmospheric component of the mpi-m earth system model: Echam6. *Journal of Advances in Modeling Earth Systems*, 5(2):146–172, 2013.

[SIL$^+$15] M. J. Schulte, M. Ignatowski, G. H. Loh, B. M. Beckmann, W. C. Brantley, S. Gurumurthi, N. Jayasena, I. Paul, S. K. Reinhardt, and G. Rodgers. Achieving exascale capabilities through heterogeneous computing. *IEEE Micro*, 35(4):26–36, July 2015.

[SKRC10] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Mass storage systems and technologies (MSST), 2010 IEEE 26th symposium on*, pages 1–10. IEEE, 2010.

[SLL01] Jeremy G Siek, Lie-Quan Lee, and Andrew Lumsdaine. *The Boost Graph Library: User Guide and Reference Manual, Portable Documents*. Pearson Education, 2001.

[SMM$^+$12] Kento Sato, Naoya Maruyama, Kathryn Mohror, Adam Moody, Todd Gamblin, Bronis R. de Supinski, and Satoshi Matsuoka. Design and modeling of a non-blocking checkpointing system. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 19:1–19:10, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[SOR$^+$03] R. K. Sahoo, A. J. Oliner, I. Rish, M. Gupta, J. E. Moreira, S. Ma, R. Vilalta, and A. Sivasubramaniam. Critical event prediction for proactive management in large-scale computer clusters. In *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '03, pages 426–435, New York, NY, USA, 2003. ACM.

[Ste03] Luciano A Stertz. Readable dirty-bits for ia64 linux. *Internal requirement specification, Hewlett-Packard*, 2003.

[STHE11] Petter Svärd, Johan Tordsson, Benoit Hudzia, and Erik Elmroth. High performance live migration through dynamic page transfer reordering and compression. In *IEEE 3rd International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 542–548, 2011.

[Top] Top 500 supercomputer. `http://top500.org/`.

[TSKK09] H. Takizawa, K. Sato, K. Komatsu, and H. Kobayashi. Checuda: A checkpoint/restart tool for cuda applications. In *Parallel and Distributed Computing, Applications and Technologies, 2009 International Conference on*, pages 408–413, December 2009.

[Tsu] Tsubame 2.0 - monitoring portal. `http://mon.g.gsic.titech.ac.jp/`.

[UNR$^+$05] R. Uhlig, G. Neiger, D. Rodgers, A. L. Santoni, F. C. M. Martins, A. V. Anderson, S. M. Bennett, A. Kagi, F. H. Leung, and L. Smith. Intel Virtualization Technology. *Computer*, 38(5):48–56, May 2005.

[Vai94] Nitin Hemant Vaidya. *A case for multi-level distributed recovery schemes.* Texas A & M University, Computer Science Department, 1994.

[Vai95a] Nitin H. Vaidya. A case for two-level distributed recovery schemes. In *Proceedings of the 1995 ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '95/PERFORMANCE '95, pages 64–73, New York, NY, USA, 1995. ACM.

[Vai95b] Nitin H Vaidya. *On checkpoint latency.* Texas A & M University, 1995.

[Vai97] Nitin H. Vaidya. Impact of checkpoint latency on overhead ratio of a checkpointing scheme. *IEEE Trans. Comput.*, 46(8):942–947, August 1997.

[VBG$^+$10] M. Valiev, E.J. Bylaska, N. Govind, K. Kowalski, T.P. Straatsma, H.J.J. Van Dam, D. Wang, J. Nieplocha, E. Apra, T.L. Windus, and W.A. de Jong. Nwchem: A comprehensive and scalable open-source solution for large scale molecular simulations. *Computer Physics Communications*, 181(9):1477 – 1489, 2010.

[VCJC$^+$13] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. Polyhedral parallel code generation for CUDA. *ACM Trans. Archit. Code Optim.*, 9(4):54:1–54:23, January 2013.

[Vir05] AMD Virtualization. Secure Virtual Machine Architecture Reference Manual. *AMD Publication*, 2005.

[VJO$^+$14] O. Villa, D. R. Johnson, M. Oconnor, E. Bolotin, D. Nellans, J. Luitjens, N. Sakharnykh, P. Wang, P. Micikevicius, A. Scudiero, S. W. Keckler, and W. J. Dally. Scaling the power wall: A path to exascale. In *SC14: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 830–841, Nov 2014.

[VMHR11] Manav Vasavada, Frank Mueller, Paul H. Hargrove, and Eric Roman. Comparing different approaches for incremental checkpointing: The showdown. Linux'11: The 13th Annual Linux Symposium, 2011.

[VTHB18] Marc-André Vef, Vasily Tarasov, Dean Hildebrand, and André Brinkmann. Challenges and solutions for tracing storage systems: A case study with spectrum scale. *ACM Trans. Storage*, 2018.

[WHV$^+$95] Yi-Min Wang, Yennun Huang, Kiem-Phong Vo, Pe-Yu Chung, and Chandra Kintala. Checkpointing and its applications. In *Fault-Tolerant Computing, 1995. FTCS-25. Digest of Papers., Twenty-Fifth International Symposium on*, pages 22–31. IEEE, 1995.

[WMES12] Chao Wang, Frank Mueller, Christian Engelmann, and Stephen L. Scott. Proactive process-level live migration and back migration in HPC environments. *J. Parallel Distrib. Comput.*, 72(2):254–267, 2012.

[Wol10a] Michael Wolfe. Implementing the pgi accelerator model. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, GPGPU-3, pages 43–50, New York, NY, USA, 2010. ACM.

[Wol10b] Katinka Wolter. *Stochastic Models for Fault Tolerance: Restart, Rejuvenation and Checkpointing.* Springer Publishing Company, Incorporated, 1st edition, 2010.

[XJFH11] Wen Xia, Hong Jiang, Dan Feng, and Yu Hua. SiLo: A Similarity-Locality based Near-Exact Deduplication Scheme with Low RAM Overhead and High Throughput. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2011.

[XLTL10] Xinhai Xu, Yufei Lin, Tao Tang, and Yisong Lin. Hial-ckpt: A hierarchical application-level checkpointing for cpu-gpu hybrid systems. In *Computer Science and Education (ICCSE), 2010 5th International Conference on*, pages 1895–1899, August 2010.

[YG07] Jing Yu and Maria Jesus Garzaran. Compiler optimizations for fault tolerance software checking. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, PACT '07, pages 433–433, Washington, DC, USA, 2007. IEEE Computer Society.

[YWGK06] Lamia Youseff, Rich Wolski, Brent C. Gorda, and Chandra Krintz. Evaluating the performance impact of xen on MPI and process execution for HPC systems. In *Proceedings of the 1st International Workshop on Virtualization Technology in Distributed Computing (VTDC@SC)*, 2006.

[ZLP08] Benjamin Zhu, Kai Li, and R. Hugo Patterson. Avoiding the Disk Bottleneck in the Data Domain Deduplication File System. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST)*, pages 269–282, 2008.