SEVENTH FRAMEWORK PROGRAMME

Research Infrastructures

FP7-ICT-2013-10

DEEP-ER

DEEP Extended Reach

Grant Agreement Number: 610476

D5.1

Checkpointing architecture design

Approved

Version: 2.0

Author(s): V. Beltran (BSC), J. Morillo (BSC)

Contributor(s): M. Cintra (Intel), N. Eicker (JUELICH), T. Moschny (ParTec)

Date: 21.10.2014

Project and Deliverable Information Sheet

DEEP-ER Project	**Project Ref. №:**	610476	
	Project Title:	DEEP Extended Reach	
	Project Web Site:	http://www.deep-er.eu	
	Deliverable ID:	D5.1	
	Deliverable Nature:	Report	
	Deliverable Level: PU*	**Contractual Date of Delivery:** 31 / March / 2014	
		Actual Date of Delivery: 31 / March / 2014	
	EC Project Officer:	Panagiotis Tsarchopoulos	

* - The dissemination level are indicated as follows: **PU** – Public, **PP** – Restricted to other participants (including the Commission Services), **RE** – Restricted to a group specified by the consortium (including the Commission Services). **CO** – Confidential, only for members of the consortium (including the Commission Services).

Document Control Sheet

Document	**Title:**	Checkpointing architecture design	
	ID:	D5.1	
	Version:	2.0	**Status:** Approved
	Available at:	http://www.deep-er.eu	
	Software Tool:	Microsoft Word	
	File(s): DEEP-ER_D5.1_Checkpointing_architecture_design_v2.0-ECapproved		
Authorship	**Written by:**	V. Beltran (BSC), J. Morillo (BSC)	
	Contributors:	M. Cintra (Intel), N. Eicker (JUELICH), T. Moschny (ParTec)	
	Reviewed by:	M. Rahn (FHG-ITWM), J.Kreutz (JUELICH)	
	Approved by:	BoP/PMT	

Document Status Sheet

Version	Date	Status	Comments
1.0	31/March/2014	Final version	EC submission
2.0	21/October/2014	Approved	EC approved

Document Keywords

Keywords:	DEEP-ER, HPC, Exascale, Resiliency, OmpSs, Checkpoint/Restart

Table of Contents

List of Figures

List of Tables

Executive Summary

This deliverable describes the software architecture that will be developed in the context of the DEEP-ER project to achieve a resiliency level that can effectively scale out on Exascale systems. These systems will require not only stronger resiliency techniques than the current ones but also more flexible and malleable approaches to deal with the heterogeneous nature of DEEP and DEEP-ER proposals.

The resilience architecture will be based on user-based checkpoint/restart techniques that provide a high level of resiliency and are the most cost-effective in terms of I/O requirements. However, this resiliency and I/O efficiency comes at the cost of increasing the applications complexity. Current user-based checkpoint/restart techniques are also too rigid and coarse-grained, as any kind of error on any node requires a full restart of the whole application from the last valid checkpoint. Besides, the offload features provided by OmpSs pose additional challenges to traditional resiliency techniques. Hence, only the mandatory improvement of the I/O performance and scalability of current checkpoint/restart libraries will not be enough to fulfill the resilience need of the DEEP-ER Prototype, which requires new resiliency features to isolate partial failures of the system without always requiring a full application restart.

To overcome the previous issues, a conventional user-based checkpoint/restart API will be complemented with novel OmpSs task-based checkpoint/restart techniques, which will result in a more resilient, fine-grained and flexible architecture. The hierarchical combination of both checkpoint/restart approaches will enable the development of malleable applications that can better isolate and recover from partial system failures. To that end, ParaStation MPI will also be extended to detect, isolate and clean up failures of offloaded tasks, which will be then independently restarted, avoiding the need of a full application recover. Moreover, both user-based and task-based checkpoint/restart techniques will be adapted to take advantage of the advanced I/O technologies and storage hierarchy provided by WP4 and WP3 respectively. Finally, a failure model will be developed to optimise polices that determine for each application the frequency, redundancy level and storage-location of each checkpoint to optimise a given goal, such as application execution time or system energy consumption. This model will take into account the probability and type of failure of the main hardware and software components, the performance of the user-based and task-based checkpoint/restart implementations on the DEEP-ER hardware architecture, as well as, the application specific characteristics.

1 Introduction

The objective of this work is to design and implement a software architecture that can cope with the requirements of future Exascale systems in terms of resiliency. State-of-the-art techniques such as user-level checkpoint/restart [**ADAMMOODY10**] [**KENTO12**] not only lack the scalability required for an Exascale system but also the flexibility to fit heterogeneous systems such as DEEP and DEEP-ER prototypes. To address both issues, we propose a novel resiliency architecture that combines user-based and task-based checkpoint/restart techniques to improve the flexibility, scalability and overall system resiliency. This report describes the resiliency architecture defined on task 5.1, which will be implemented in tasks 5.2, 5.3 and 5.4 in collaboration with WP3 and WP4.

1.1 Resiliency on Exascale systems

Current large-scale data centres already show hardware-failure rates that require a resilient software layer [**BSCHROEDER09**] [**EL-SAYED13**] [**AHWANG13**]. Despite the relatively high resiliency of individual components, Exascale systems are expected to exacerbate this trend [**FCAPELLO09**]. Hardware vendors have already realised that hardware-only solutions will not scale, so cooperation between hardware and software layers is becoming mandatory. This hardware/software cooperation will be especially important to deal with the expected increase of transient errors that today always results in full node failure.

	Type of errors		Description
Detected	Corrected Error (**CE**)		A error detected and corrected by Hardware
	Uncorrected Fatal Error (**UC**)	Transient (**TUC**)	Fatal hardware failure. System must restart to be operational again
		Permanent (**PUC**)	Fatal hardware failure. System must be repaired to be operational again.
	Uncorrected Recoverable Error (**UCR**)	**SRAR**	Some data was corrupted and it has been consumed. **Action** is **required**.
		SRAO	Some data is corrupted but it has not been consumed. **Action** is **optional**.
		UCNA	Some data is corrupted but it has not been consumed and the system may continue to operate. **No action** is required.
Undetected	Benign Error (**BE**)		Some data is corrupted without any error being logged but the contents are correct.
	Silent Data Corruption (**SDC**)		Some data is corrupted without any error being logged but the contents are not correct.

Table 1: Classification of hardware errors

Table 1 shows the classification of errors in terms of the recovery actions that the hardware/software can take on a Linux system. Errors are first classified as detected or undetected, the latter ones may be further classified as Benign Errors (BE) if they do not affect the normal operation of the system or Silent Data Corruption (SDC) that will led to erroneous computations or even a whole system crash. DEEP-ER resiliency techniques will

focus mainly on detected errors, but some techniques to minimise the number of undetected errors will be also explored. Inside the detected errors there are three subcategories: Corrected Errors (CE), Uncorrected Fatal Errors (UC) and Uncorrected Recoverable Errors (UCR). The CE are detected and transparently corrected by the hardware. An illustrative example of a CE is an arbitrary flip of a bit in a memory location that is detected and transparently corrected by the hardware ECC. The second subcategory of detectable errors is the Uncorrected fatal Error (UC). These errors are detected by the hardware but are too severe to be corrected by hardware or software (ex: CPU failure), so the only option is to shutdown the system as soon as possible to avoid further damages or data corruptions. If the error is transient the system will be operational after a reboot, on the other hand, if the error is permanent the system must be fixed to be operational again. The last subcategory groups the Uncorrected Recoverable Errors (UCR). These errors are identified by the hardware but cannot be automatically fixed. An example of this type of error is a multi-bit flip in a memory location that can be detected, but not corrected, by the ECC hardware. UCR errors are further classified into SRAR, SRAO and UCNA. The SRAR errors require an action from the software layer to be able to continue with the system execution, which may consist in killing the application that has triggered this error. The OS layer can usually isolate and handle SRAO and UCNA errors, so they are virtually transparent to the applications running on the system.

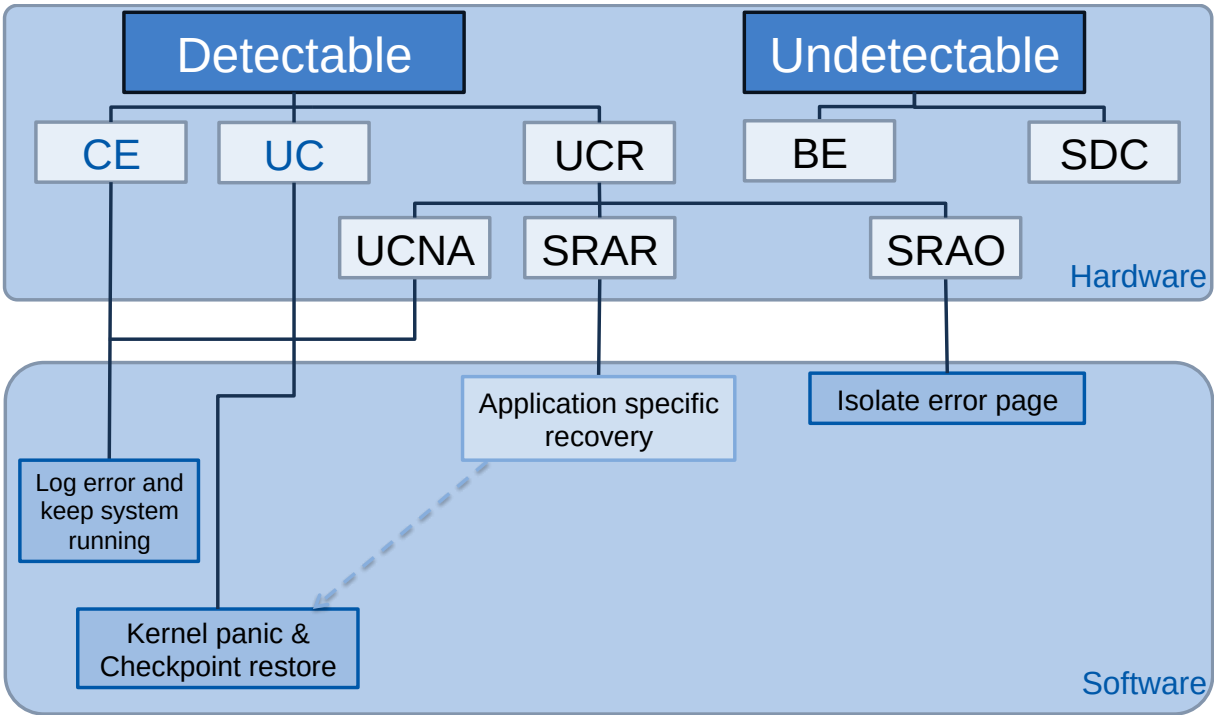

Figure 1: Standard recovery actions on Linux OS using the Machine Check Architecture (MCA)

Hardware vendors like Intel have realised that, even on current systems, hardware techniques are not enough to provide a high level of resiliency. To mitigate this problem last generation Xeon Processors [**INTELE7**] include several resilient features that enables a close interaction of hardware and software layers to improve system resiliency. The Machine Check Architecture (MCA) [**INTELMCA**] is a framework that enables all the software layers, from the OS to user applications to better handle some types of errors. Figure 1 shows a diagram with the different types of errors and recovery actions available on the Linux OS.

The MCA hardware notifies the OS each time an error is detected. For CE the kernel just logs the type and location of the error. This information can be used to predict future failures and also to implement early retirement of memory pages with higher probability of failure. For UC errors the only option is to shutdown the system as soon as possible, to avoid any further damage or data corruption. UCR errors cannot be solved by the hardware, but are not critical enough to require a system shutdown. For UCNA and SRAO subtypes the OS can handle the errors just logging the fault for the former or isolating the error for the latter. For SRAR the OS layer does not have enough information to handle the error so cooperation from the application running is required. That means that to improve the resiliency of the system applications have to be "Recovery Aware". On Linux, this means that applications have to provide a signal handler with an appropriate code to handle errors on its assigned resources. If the error cannot be handled the application is then killed.

1.2 Hardware heterogeneity and application requirements

The DEEP/DEEP-ER project is intended to develop a novel supercomputer architecture that departs from existing approaches. While traditional supercomputers are homogeneous at the cluster level but can be heterogeneous inside each node by using some kind of accelerator, our approach targets a heterogeneous supercomputer composed of Cluster Nodes (CN) and Booster Nodes (BN). The CNs are based on a traditional multi-core Xeon processor while the BNs are based on the many-core Xeon Phi processor (KNC on DEEP and KNL on DEEP-ER). To make the most of this heterogeneous cluster architecture, OmpSs programming model and ParaStation MPI have been extended to ease the collectively offload of MPI kernels from the CNs to the BNs and vice-versa, improving the flexibility and malleability of MPI applications. Existing checkpoint/restart techniques are not prepared to cope with the challenges that this flexible programming model introduces. Hence, a novel resilient software architecture is required not only to scale on Exascale systems, but also, to cope with a heterogeneous cluster architecture. Ideally, current techniques should be extended to support more fine-grained checkpoint/restart techniques that can handle most transient or soft-errors without requiring a whole application recovery and can also isolate partial cluster failures, only requiring the recovery of the affected part of the application.

1.3 DEEP-ER approach to resiliency

The resiliency architecture of DEEP-ER will combine well known user-based checkpoint restart techniques with novel task-based checkpoint restart techniques, described on Sections 3 and 4 respectively. These techniques have to cover both UC and UCR failures, as well as, fulfil the application requirements presented on Section 2. However, just applying both techniques in isolation will not be enough to cope with the heterogeneous nature of DEEP-ER Prototype nor the flexible programming model provided by the DEEP offload. With the DEEP Offload it is possible to easily offload a large MPI kernel from a set of nodes to another set of nodes, which can be seen as a long running task. For this kind of long running task it is not enough to save the inputs and restart from the beginning in case of failure, because a lot of work can potentially be lost. To address this situation we plan to extend user-level checkpoint so it can be optionally used inside long running tasks in a hierarchical way. Hence, the whole application will not be limited to checkpoint and restart in only one location, supporting up to one checkpoint/restart for each offloaded/long running tasks. The idea behind this hierarchical integration of user-level and task-based checkpoint restart is explained in Section 5. Finally, a failure model to optimise both user-level and task-based

checkpoint restart is mandatory. Section 6 presents a model that given the expected probability of each class of error will determine the optimal frequency and location (local, buddy or global) of the checkpoint/restart techniques to minimise application execution time or total system energy consumption.

2 Application requirements

In this section we summarise the behaviour of the DEEP-ER applications in terms of their reliability requirements. A more complete description of the applications can be found in deliverable D6.1.

- Astronomy Application (owner: ASTRON)

This application has real-time requirements and a streaming data model. High availability is of foremost importance for this application. This translates into a requirement for fast restarts of the application. However, the streaming real-time behaviour means that there is no need to restart from a checkpoint state. For this reason, CR support in DEEP-ER is not expected to be beneficial for this application. However, task-based resiliency might be considered to provide error isolation and fast application recovery features.

- BSIT Full Waveform Inversion (owner: BSC)

This application performs iterative computations on a large mutable dataset, where the entire dataset resulting from each iteration is saved for the next iteration. This also serves as an implicit CR mechanism for this application and could leverage the proposed CR mechanism of DEEP-ER. However, there is currently no mechanism for checkpointing intermediate state during each iteration. The amount of data that must be checkpointed depends heavily on the type of cell used and varies from tens of GB for conventional grid cells to hundreds of GB for FSG cells. Given these characteristics, there seems to be a benefit for CR at intermediate points in the computation, but this must not slow down the critical path execution despite the large amounts of data that need to be checkpointed.

- High temperature superconductivity with TurboRVB (owner: CINECA)

This application performs Quantum Monte Carlo simulations. Restart files are generated periodically during the simulation. These also serve as an implicit CR mechanism for this application and could leverage the proposed CR mechanism of DEEP-ER. Moreover, the frequency of restart files is a user parameter and it is possible that a more efficient CR mechanism may allow for more frequent checkpointing. Also, the division of work to semi-independent and well-defined “walker” tasks in this application make it amenable to use OpenSs’s task run-time. This opens the possibility to also use the proposed task-based CR approach in addition to user-level CR.

- Assessment of Human Exposure to Electromagnetic Fields (owner: INRIA)

This application consists of a Discontinuous Galerkin Time-Domain (DGTD) solver of the Maxwell-Debye system in 3D. Functionality to save checkpoint files at some user-defined frequency of iterations of the main loop is already present in the application. This mechanism could leverage the proposed CR mechanism of DEEP-ER. Higher frequencies of such checkpoints could be attained with more efficient CR mechanism. Currently the amount of data checkpointed ranges from a few GB for the “head” input to tens of GB for the “Louis” input. Other, larger, inputs are currently being investigated.

- iPic3D (owner: KULeuven)

  This application is also an iterative solver of electromagnetic interactions. Program state is divided into field data and particle data. The first currently amounts to a few GB but the latter amounts to hundreds of GB. The state of fields and particles are stored periodically to files (with fields being stored more often due to their smaller size), which serves as an implicit CR mechanism for this application. This mechanism could leverage the proposed CR mechanism of DEEP-ER. In particular, a more efficient CR mechanism could allow for more frequent checkpoints of particle data.

- SeisSol (owner: BADW-LRZ)

  This application is another variation of Discontinuous Galerkin solver and uses Arbitrary high order DERivatives (ADER) for time discretisation. There is currently no built-in restart mechanism, but CR could be easily done at some number of iterations of the outer loop. The amount of data varies depending on the order of elastic case used, from a few GB for 4^{th} order to some tens of GB for 6^{th} order. Again, integration and leveraging of the CR mechanism of DEEP-ER is possible.

- Chroma (owner: UREG)

  This is a Lattice QCD application and if Markov chains are simulated, intermediate results are written out. The Markov chain can then be restarted from any of these intermediate results, which provides an implicit CR mechanism for this application. This mechanism could leverage the proposed CR mechanisms of DEEP-ER. The amount of data currently stored at each step is in the order of a few tens of GB. No finer-grained or more frequent checkpointing appears to be required for this application.

Most applications will be modified to use the DEEP Offload mechanism to make the most of the heterogeneous nature of the DEEP/DEEP-ER architectures. Hence, current CR methods will have to be revisited and extended with the resiliency techniques presented in Section 5.

3 User-based checkpoint/restart

User-based checkpoint/restart (CP/RS) is a well established technology in order to increase the resiliency of large-scale HPC applications [**ADAMMOODY10**] [**KENTO12**]. The key idea behind it is to store all the information required to re-establish an application run in a distinct state to a persistent storage. In the rare case of a failure this information is used to restart the application and to restore its internal state that was present at the time of writing the checkpoint. A central aspect of this approach is to find the optimal balance between the amount of checkpoints written, the probability of requiring the checkpoint for restart and the overhead introduced by writing the checkpoint.

Of course, the cost of writing a single checkpoint strongly depends on where it is written to. While writing the checkpoint to global file system maximises the probability that it might be usable after an incident, it comes with a high costs since bandwidth to such file systems is typically limited. To the other extreme, storing checkpoints at local disks or even to a local file-system located in memory is inherently scalable and comes at a lower price due to the higher bandwidth. Nevertheless, the overall checkpoint might be unusable since the portion stored to the local disk of a failed node might be inaccessible. Therefore, the DEEP-ER approach foresees to have a multi-level checkpointing strategy combining checkpoints

written to various locations (and therefore different cost) providing a range of ratios of access-costs vs. dependability. It will include at least checkpoints written to the global file system and local checkpoint, but further strategies are under investigation: On the one hand local checkpoints might be stored to so-called buddy nodes, i.e. partner nodes that reduce the probability that a portion of a checkpoint might become inaccessible. On the other hand caching techniques provided by FhGFS might be used to keep the checkpoints written to the global file system in the local cache or to even create backup copies in the cache of a second node acting then as a buddy.

Traditional user-based CP/RS techniques are from an implementation point of view distinct to each application code. Frequently the actual I/O routines are derived from output of simulation data that might be used to store results or for visualisation purposes. They might be enriched by additional data required to restart the code at a given point in time. This data is typically accessed via regular POSIX I/O in a process-local fashion, i.e. each of the participating processes writes its own piece of data to a separate file. Alternatively, parallel I/O techniques like MPI-IO or its database counterpart parallel HDF5 are used. More advanced approaches may also use ADIOS [1] in order to reach a better parallel I/O performance. All of these approaches share the characteristic, that their implementation of CP/RS is neither easily portable nor applicable to any other user code.

Nevertheless, most of the functionality of CP/RS is similar, if not common, for most applications. This includes decisions on when to write checkpoints, where to store checkpoint, how to find valid checkpoint during restart, all sort of meta-data handling, etc. Therefore the provisioning of corresponding functionality in a centralised fashion will release application-developers from a severe burden. Furthermore, this approach allows taking decisions on checkpoint frequency, target, etc. in a centralised way without the requirement of touching each and every application individually.

A good example for this type of CP/RS functionality is provided by the SCR [**ADAMMOODY10**] library. It is a good example on how to offer a common interface to the user code. At the same time it tries to hide the underlying complexity of data transfers to or from a certain storage level or meta-data handling from the user. WP5 plans to start the design of a user-based checkpoint restart approach inspired by or based on this library. A final decision is outstanding at the moment and under investigation. Nevertheless, it has to meet the requirements of user codes under investigation in WP6 as well as the target platform with its hierarchic storage architecture.

The library should offer an interface similar to SCR. This includes functions to ask whether a checkpoint is necessary, to store pre-defined datasets to a checkpoint and to recover a dataset from a checkpoint back into the user code. The decision, when a checkpoint is required, might use a purely time-based approach. In this case the user might define a percentage of runtime which is affordable for creating checkpoint. Alternatively, an iteration-based approach might be chosen. Here the user has to define a phase of the iteration-cycle when it's appropriate to create a checkpoint. In addition, different checkpoint frequencies for different storage levels are foreseen, with respect to transfer speed and size. In the course of the project a detailed failure model will be developed fixing the various parameters required for these decisions. Section 6 sketches an early draft of this model.

3.1 Integration with scalable I/O technologies

As already sketched, a key challenge for an efficient implementation of CP/RS functionality is a powerful implementation of I/O capabilities. It will enable application developers to dump the actual data of the checkpoint to the storage in an optimised way. This aims for reducing the overhead introduced by the repeated storing of checkpoints as much as possible.

DEEP-ER will follow two paths in order to implement such functionality, SIONlib and FhGFS. Both principle technologies are developed and improved within WP4 in order to enable applications to fulfil their regular I/O requirements. Of course, CP/RS has special requirements beyond that. This section briefly describes both technologies and sketches the extensions that are necessary for a seamless integration with the resiliency functionality developed within WP5.

3.1.1 SIONlib (WP4)

SIONlib [**Frings; Wolf; Petkov (2009)**][**Freche; Frings; Sutmann (2010)**] is motivated by task-local I/O. It aims to increase the efficiency of this approach to I/O on parallel machines at very large scale with the focus on omitting single files per process. Instead it offers an API to the application developer that provides the perspective of single output files per task while at the same time utilises single or few files on a parallel file system. The emphasis of SIONlib is on optimising the I/O performance with a minimum amount effort for the user. A more detailed description of SIONlib is provided with D4.1.

Since the original motivation of SIONlib stems from the need of optimisation of writing checkpoints for large-scale applications, this choice appears to be natural. Even though the original target storage of SIONlib was a global file system in the meantime it has the capability to utilise local storage, too. This allows writing both, local and global checkpoints, using the same interface. In this context it has to be pointed out that local checkpoints are still initiated in a global way. I.e. even though they are stored locally, all processes being part of the parallel application store them in a coordinated effort leading to a globally consistent checkpoint once all local operations are finished.

For local checkpoints SIONlib is able to handle meta-data information in a transparent way utilising a global file system. Extensions might be required in order to enable SIONlib to access local checkpoints after an application got struck by an incident forcing it to be re-started at a different set of resources. For that, WP5 aims to implement a corresponding functionality based on ParTec's ParaStation process management allowing for a remote-access of local checkpoint data.

In order to allow for buddy checkpointing via SIONlibs interface, additional functionality is required. For that, SIONlibs capability to collect data of several MPI-processes before it is written to the file-system that was originally implemented to tackle the small-I/O problem of HPC will be abused. Using the same functionality data will be forwarded to the buddy node and written to the local file system there. Again, SIONlib will handle meta-data information in a transparent way using a global file system.

While at a first glance the type of functionality provided by SIONlib seems to fit directly into the file system in a natural way, we decided to follow this line of development anyhow for different reasons. First of all the relatively simple structure of SIONlib provides the possibility to undertake experiments with significantly lower effort and much higher flexibility. By that it is possible to identify and evaluate approaches that might be integrated into the actual file

system in a later stage. Furthermore SIONlib offers the opportunity the implement such ideas in a file system agnostic way leaving the possibility to integrate them with other file systems later on.

3.1.2 FhGFS global file system + per node cache (WP4)

FraunhoferFS (FhGFS) [**FhGFS website**] is the high-performance parallel file system from the Fraunhofer Competence Center for High Performance Computing. Its distributed metadata architecture has been designed to provide the scalability and flexibility that is required to run today's most demanding HPC applications. Beyond performance it takes emphasis on ease of use and administration. A more detailed description of FhGFS is given in D4.1.

DEEP-ER's user-based CP/RS functionality aims to utilise FhGFS caching capabilities in combination with its ability to mirror data stored to it. In order to disburden the file system's storage servers from the massive I/O load created by the synchronous write of checkpoint data in large-scale applications, file system caches managed by FhGFS on the compute nodes shall be used to keep checkpoints local. These caches will utilise node-local storage devices and offer the possibility to seamlessly integrate novel storage devices like NVM discussed later on. Since most of the checkpoint will never be used and will be annulled as soon as the next checkpoint was written, this approach increases the performance of the system significantly. Even writing checkpoints to the central storage servers might facilitate the cached version by moving them asynchronous to the actual write operation of the user-based checkpointing later on in some background thread.

In order to store buddy checkpoints WP5 aims to utilise FhGFS mirroring capabilities. Mirroring local caches to the cache of a buddy node might even be handled in an asynchronous way in the background while the application already started to continue its normal execution. Of course, this might increase the complexity of the decision, if a checkpoint is actually usable during a restart since buddy checkpoints might still be in an inconsistent state even though writing the local checkpoints was finished successfully.

3.2 Exploration of emerging hardware (WP3)

DEEP-ER aims to integrate innovative memory technologies into the system, too. Both technologies, non-volatile memory (NVM) and network-attached memory (NAM) promise to be beneficial for the CP/RS approach of WP5. This section briefly motivates how to gain from these technologies and how to integrate them into the concept presented within this document.

3.2.1 Non-volatile memory (NVM)

Writing local checkpoints – either explicitly to a local filesystem via some functionality described in the context of SIONlib or implicitly via FhGFS caching feature – of course requires local storage devices. Traditionally this utilises local harddisks residing in the compute-nodes of the HPC system or their main memory via an in-memory file system. While the latter might reduce the capacity of the memory available for actual applications significantly, harddisks provide typically only poor bandwidth increasing the costs of a checkpoint significantly. A good alternative might be provided by non-volatile memory devices based on flash or similar technologies in the future.

DEEP-ER will explore the potential of the novel NVMe technology. Currently such devices are realised as PCIe cards and might be accessed as a traditional block-device or via a novel API allowing a more fine-grained access. A detailed description of its functionality and capabilities is provided in D3.1.

In the context of CP/RS an obvious use-case for NVMe devices is to use it to implement a local file-system. That might be utilised either via SIONlib to hold local checkpoints and the ones coming from the buddy node or as the local cache locations for the global FhGFS. Both approaches only require minor modifications compared to the use of traditional harddisks.

3.2.2 Network-attached memory (NAM)

Network-attached memory is a promising technology that might revolutionise the use of distributed memory machines in the near future. The key idea is to facilitate memory within the network without having general-purpose compute- or execution-capabilities aside of it. A NAM-device basically will consist of a network interface card (NIC) with some memory attached. It will be accessed via standard RDMA operations that have to be provided by the interconnect fabric. A detailed description of this technology is provided in D3.1.

For user-based CP/RS NAM is a natural choice in order to store buddy checkpoints. Of course, even though such devices might be distributed within future HPC systems, their capacity might be limited. This will require a significant compression of the checkpoint data required to store to such a device. Nevertheless, the nature of the expected failure-modes provides the opportunity to realise such compression of the date to be actually stored as explained now.

In a failure-case expected to be typical for Exascale systems, only one or few buddy checkpoints will be required since most of the local checkpoints will still be accessible. Thus, it will be sufficient to just store parity[1] information to the NAM. For that DEEP-ER explores the possibility to implement the NIC within an FPGA-device which would allow shifting the effort to compute parity information to the NIC. For that, the compute-nodes sharing a NAM-device to hold their parity information just have to send their local checkpoints to the corresponding NAM. The NIC-section of this device will then compute the corresponding parity information on the fly while receiving the checkpoints and writing it to the local memory.

4 Task-based checkpoint/restart

This Section describes how OmpSs dataflow nature can be used together with the Machine Check Architecture (MCA) hardware to transparently improve application resiliency. The key idea is to use automatic and lightweight in-memory checkpoint of task inputs, so in case of a task failure the runtime can isolate the error and re-execute the affected task.

4.1 OmpSs programming model

OmpSs is a data-flow programming model that extends OpenMP with pragma annotations to support tasks. These annotations are interpreted by the Mercurium source-to-source

[1] Parity shall be seen as a more general form of redundant information to be stored to the NAM. This does not restrict to simple XOR parity but may also include more complex codes like Reed-Solomon, which would allow recovering from losing multiple local checkpoints.

compiler, which emits calls to Nanos++, the runtime system. Nanos++ uses the information provided by user annotations to dynamically build at runtime a task dependency graph, which is used to schedule tasks in a data-flow way.

```
#pragma omp task inout ([TS][TS]A)
void spotrf (float *A);

#pragma omp task input ([TS][TS]T) inout ([TS][TS]B)
void strsm (float *T, float *B);

#pragma omp task input ([TS][TS]A,[TS][TS]B) inout ([TS][TS]C)
void sgemm (float *A, float *B, float *C);

#pragma omp task input ([TS][TS]A) inout ([TS][TS]C)
void ssyrk (float *A, float *C);

void Cholesky( float *A ) {
   int i, j, k;
   for (k=0; k<NT; k++) {
      spotrf (A[k*NT+k]) ;
      for (i=k+1; i<NT; i++)
         strsm (A[k*NT+k], A[k*NT+i]);
      // update trailing submatrix
      for (i=k+1; i<NT; i++) {
         for (j=k+1; j<i; j++)
            sgemm( A[k*NT+i], A[k*NT+j], A[j*NT+i]);
         ssyrk (A[k*NT+i], A[i*NT+i]);
   }
}
```

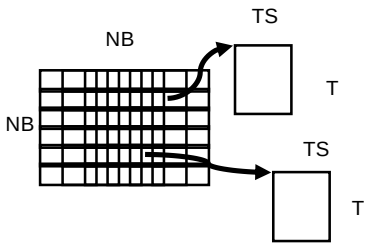

Figure 2: OmpSs version of Cholesky decomposition

Figure 2 shows a sequential implementation of a Cholesky decomposition that has been augmented with OmpSs pragmas to specify *in*, *out* and *inlout* parameters of functions *spotrf, strsm, sgemm* and *ssyrk.* The notation is straightforward, the *#pragma omp task* is used before a function declaration to mark it as a task, and the *input, output or inout([size]var_name)* clauses to specify the size of the input and output variables of each function. This additional information is parsed and analysed by the Mercurium compiler to convert the original source file with annotations to a new source file with the original code transformed to call to the Nanos++ runtime system. For each plain call to the annotated functions on the original file, Mercurium will generate a call to the runtime system to create a new task. The resulting file is finally compiled by a native compiler and linked with Nanos++.

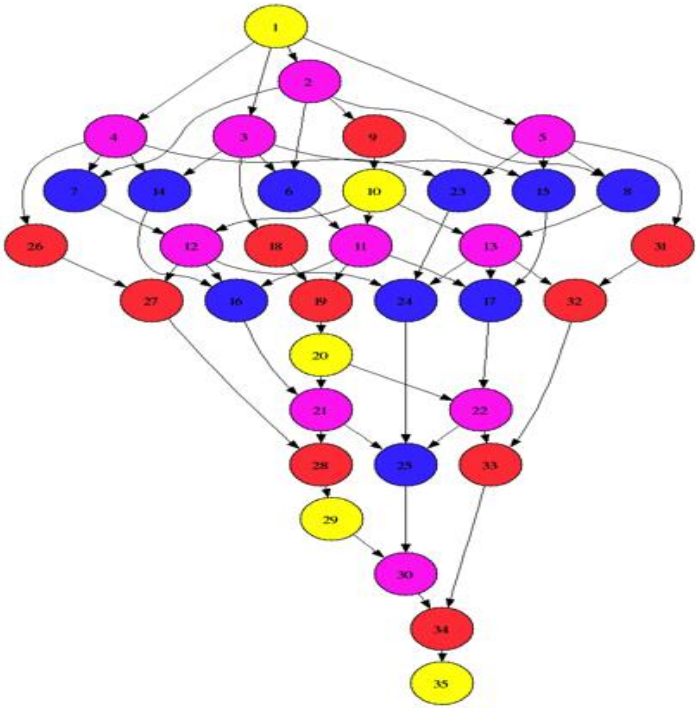

Figure 3: Task dependency-graph of Cholesky

The execution of the binary will generate a new task each time one of the previous functions were originally called. Then, Nanos++ will dynamically generate the task dependency graph at runtime, so it will not run a task until all of its dependencies are fulfilled. Figure 3 shows the task dependency graph of the previous Cholesky example. This information is used to optimally map tasks that can run in parallel on the available resources but, as explained in next Section, it can also be used to transparently improve the resiliency of any application written with OmpSs.

4.2 OmpSs task-based checkpoint/restart

OmpSs can use the dataflow information provided by the *in* annotations to automatically checkpoint in main memory the inputs of a task (see Section 4.3 for more details about the checkpoint operation), so in case of a task-failure the runtime system can automatically re-execute the affected task. To benefit from this automatic recovery mechanism tasks must be side effect free. Most computational tasks have no side effects, but the ones that perform I/O operations cannot usually use this approach. OmpSs uses the signalling mechanism provided by the MCA to handle Uncorrected Recoverable (UCR) errors. To that end, OmpSs installs a signal handler that captures any *sigbus* signal sent to the application. The information provided by the *sigbus* signal is used to first identify the memory address affected by the UCR error, which is then used to identify the tasks that are currently accessing that memory. The failing task, as well as any nested task, is immediately aborted and the runtime proceeds with the recovery procedure. First, the corrupted data is replaced with the check-pointed data and then the task is re-executed. It is worth noting that only the affected tasks are aborted and re-executed while the rest of the tasks can continue the execution unaffected. Figure 4 shows how the OmpSs checkpoint/restart extends the MCA architecture, providing a generic framework that can transparently improve the resiliency of any OmpSs application. When a UCR error occurs, the runtime will determine the precise location of the error and it will proceed with the appropriate recovery action. If the error has

occurred inside a task, this task will be immediately aborted and re-executed with the check-pointed inputs, so the rest of the tasks can continue unaffected. If the error occurs inside the runtime, a Nanox specific recovery function will try to recover from the error (for example, it may happen that the error occurs while the runtime is performing the checkpoint of the tasks inputs). If the recovery is not possible the application will terminate and the user-level checkpoint/restart mechanism will be used to recover the application execution from the last valid checkpoint. If the error is inside the application, but outside a task, the (optional) application specific recovery mechanism will be called. If the recovery action succeeds, the application will continue the execution; otherwise the application will be terminated and recovered from the last valid user-level checkpoint.

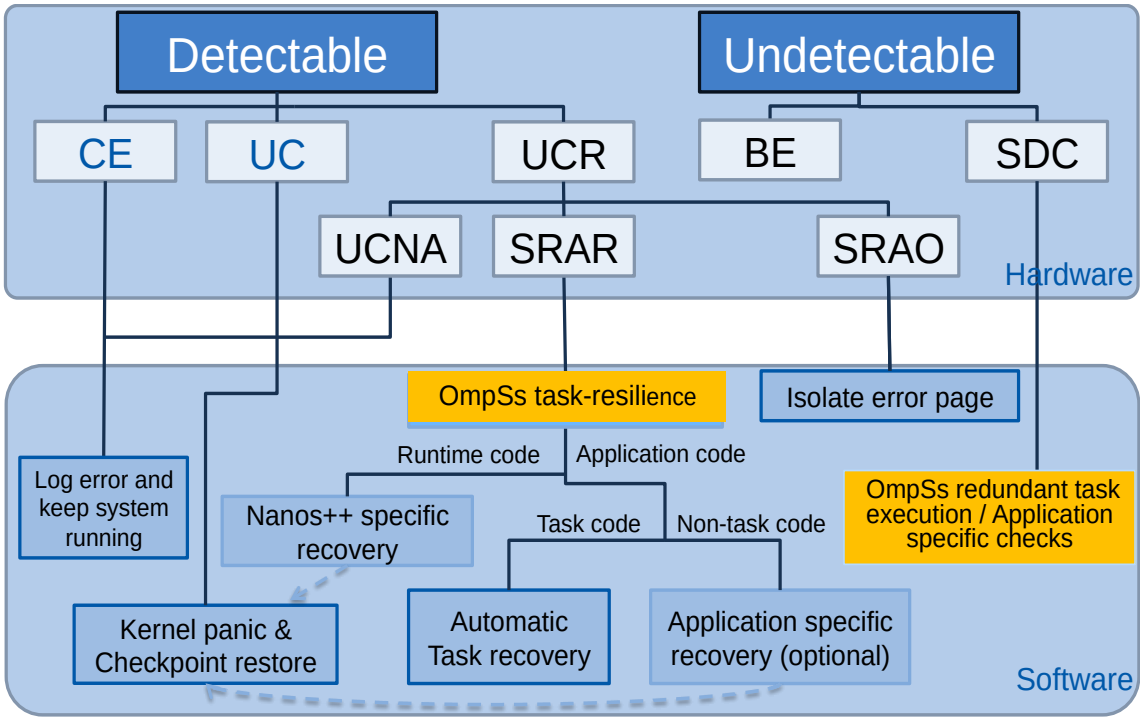

Figure 4: Extended recovery actions with OmpSs on Linux using the Machine Check Architecture (MCA)

The task-based resiliency provided by OmpSs is very lightweight and convenient compared to traditional user-based checkpoint/restart. Moreover, it can successfully handle most of the UCR errors that otherwise will led to a user-level recovery action. However, task-based checkpoint/restart only protects against UCR errors and can only be used on tasks without side-effects so it has to be used in conjunction with user-based checkpoint/restart to provide protection against a wider spectrum of errors. Section 5 explains how we plan to integrate both techniques.

4.3 Exploration of emerging hardware (WP3)

The task-based checkpoint/restart is based on check-pointing the input parameters of tasks on a free memory location. This in-memory checkpoint has the best performance but, in some situations, the memory used to store checkpoints can grow quickly, reducing the available memory for application, which may severely impact performance. The NVM devices have better performance than traditional storage systems but RAM performance is still unmatched. Moreover, task-based resiliency cannot take benefit of the persistent nature of

NVM, so check-pointing all the task inputs directly to NVM is not worth. However, NVM devices can be used to address the capacity problem of the RAM without compromising the performance of in-memory checkpoint. The key idea is reserve via a conventional *mmap* call a portion of the virtual address space proportional to the size of the system memory. Then, all the checkpoints will be stored in this region, which will be mapped to a NVM device. With this configuration the checkpoints will be a memory-to-memory operation and, when the memory usage increases, the OS will transparently swap-out some of the check-pointed data to the NVM to increase the available memory. In case of task failure the OS will automatically swap-in any required checkpoint that was stored on the NVM.

5 Integration of user-based and task-based checkpoint/restart

Coarse-grained or long running tasks are the ones that take a long time to complete. For these long running-tasks, the task-based checkpoint/restart approach is suboptimal because in case of a failure the failed task can only be recovered from the very beginning, potentially loosing a lot of computational time. This problem is exacerbated for offloaded tasks, which by its own nature are not only usually coarse-grained but also can run on a potentially large number of nodes. Thus, if one of the nodes that is running the offloaded task fails all the computations of the rest of nodes are also thrown away. To overcome these limitations we need to nicely integrate task-based and user-based checkpoint/restart techniques.

5.1 Support of DEEP offload tasks

The offload of high performance kernels written in MPI is a key feature to effectively exploit the DEEP/DEEP-ER architecture. Current MPI implementations already support the *MPI_comm_spawn(…)* primitive to dynamically create new MPI processes on arbitrary nodes. This collective operation creates an intra-communicator that connects the set of nodes that call it, with the new set of MPI processes created on the specified remote nodes. It is worth noting that this MPI primitive supports the creation of new MPI processes even on nodes with a different architecture, which makes it especially well-suited for the DEEP/DEEP-ER Architecture. However, this function can dramatically increase the complexity of an MPI application because the programmer has to coordinate and manage two or more sets of parallel MPI processes, explicitly sending the required data from side to side. This additional complexity makes this technique cumbersome or even unfeasible on large and complex applications such as the ones targeted by DEEP/DEEP-ER. To overcome the above-mentioned issues, OmpSs has been extended to support a flexible approach to offload MPI kernels from CNs to BNs. The goal is to ease, as much as possible, the mechanism to offload MPI kernels, but leveraging the current MPI infrastructure. To that end, we have designed an offload model that uses *#pragmas* to mark the *MPI-kernels* that must be offloaded. Then, the compiler and the runtime systems cooperate to transparently manage all the data transfers between the MPI processes on CNs and MPI processes on the BNs. This pragmatic approach leverages both the unmodified MPI kernels of the application and the optimised communication infrastructure provided by the MPI library. Our runtime uses under the hood the *MPI_comm_spawn(…), MPI_comm_send(), etc,* functions allocate new MPI processes and to transfer the required data from CNs to BNs and vice-versa. This approach increases the malleability of the applications and has the potential to apply advanced optimisations using the transparent directory/cache implemented on the runtime system.

From the programmer point of view, offloaded tasks are like regular task but from the implementation point of view the way to detect failures and replay the execution is very different. While for traditional tasks MCA is used to detect a failure, for offloaded tasks this responsibility is delegated to the MPI library, which is also in charge of cleaning up any failing nodes. The recovery mechanism is the same than for regular task, the task is re-executed from the very beginning from the original input. This task-based checkpoint mechanism is coarse-grained for offloaded tasks and has to be augmented with user-level checkpoint-restart techniques. The integration of both technics is explained in the next subsection.

5.2 Hierarchical integration of task-based and user-based checkpoint/restart

Task-based resiliency is more fine-grained and lightweight than user-based checkpoint/restart for regular task, but for long running tasks and offloaded tasks, it is the other way around. Task-based resiliency is too coarse-grained and must be used in conjunction with user-level checkpoint/restart techniques to improve the granularity of the checkpoints. This method combines the task-based approach with use of user-level checkpoint/restart inside the long running tasks. In this scenario, the user only needs to use the user-based checkpoint/restart to save and restore the task state not the whole program state. Thus the application of this user-based checkpoint/restart becomes more scalable, as the programmer only needs to think in a task-by-task basis, without requiring a complete understand of all the application. If a long running task fails, the runtime performs the same operations than for a regular task, but the first thing that the task does when it is re-executed is to check if there is a user-level checkpoint of the task available to fast-forward the execution to the latest execution point before the failure. We plan to implement support for multiple checkpoint images on the user-based checkpoint/restart libraries to support this mode of operation. This approach can also be used in a hierarchical way defined by the nesting level of the tasks. Only one active checkpoint per level is possible, but several nested levels will be supported. With this technique it should be possible to offload to different tasks to a two different sets of nodes, while the main MPI application is performing other computations. If one of the offloaded tasks fails, the task is re-executed and then the task itself fast-forwards the execution to the latest valid checkpoint inside the tasks, the main MPI application and the other offloaded tasks remain un-affected. The only time lost is the one between the latest checkpoint and the time of failure. On the other hand, if the main application fails, the main application and both offloaded tasks are aborted. However, once we restart the application, the user-level checkpoint will fast-forward the execution of the main MPI application to the latest valid checkpoint that will then instantiate the two offloaded tasks. The offloaded tasks will also fast-forward the execution to the latest valid checkpoint, minimising the impact of the failure. The integration of user-based and task-based checkpoint/restart improves the granularity of task-based checkpoint for long running tasks and, at the same time, it also improves the malleability and flexibility of traditional user-level checkpoint/restart, supporting offloaded task recovery to isolate partial cluster errors.

5.3 Extensions required on ParaStation MPI

5.3.1 ParaStation Global MPI

ParaStation MPI plays a central role in the DEEP programming model which the DEEP-ER project inherits to some extent. Supporting a variety of interconnects through specifically

optimised communication plugins, it is used by applications on the Cluster part as well as by highly scalable code parts (HSCP) on the Booster part of the DEEP system. The ParaStation communication library pscom transparently supports Infiniband and Extoll interconnects as well as shared memory communication. Together with an offloading mechanism and the Cluster-Booster communication, the ParaStation MPI provides a global MPI, i.e. a heterogeneous MPI implementation allowing for communication between any two nodes of the DEEP system. The offloading mechanism is implemented using the dynamic process model of MPI-2, namely MPI_Comm_spawn(). This allows an application to spawn additional processes on nodes of any hardware type and furthermore provides a means for efficient data exchange between all parts of the application.

A significant effort has been made to extend the resource management system (RMS), commonly referred to as the *batch system*, as well as the management components of ParaStation MPI, to support a heterogeneous system and to allow applications to explicitly allocate nodes of different hardware types. This can happen either *statically* (i.e. at job start) or *dynamically* (at job run time). MPI_Comm_spawn supports both cases, either placing children on pre-allocated nodes or querying the batch system for additional nodes. For any of these children, the required hardware type can be specified.

For the DEEP-ER Prototype, we plan to base the MPI implementation on the ParaStation Global MPI, exploiting the features described above and extending it to the needs of the project, namely its resiliency aspects. The required extensions are detailed in the following sections.

5.3.2 Changes in the ParaStation management daemon

In the hierarchical checkpoint/restart mechanism described in this section, groups of processes perform checkpoints in a collective and coordinated fashion. In case one of these processes stops for any reason (e.g. because the node it runs on has a hardware issue), the remaining processes of the same group have to be stopped. Then the whole group restarts based on its last, common checkpoint.

The MPI implementation should be extended to support disconnecting groups of processes so that stopping (and restarting) one group does not force all process groups related to the first group to also automatically stop. In an MPI_Comm_spawn scenario this means that parents and children are able to disconnect and stop independently.

To support checkpoint/restart, proper integration with the RMS is needed. A failing node should not end the whole batch job; instead parts of the application are to be restarted in the allocation. Restarted processes may obviously run on nodes different from the nodes they have been started on initially.

The ParaStation daemon *psid* and its plugins are the components of the ParaStation MPI responsible for starting processes and communicating with the RMS. The psid will be modified to properly handle the details of such a restart. It is e.g. important to provide stable rank numbers to as many processes as possible, to exploit the locality of checkpoint data on healthy nodes.

In case the number of failing nodes exceeds the number of nodes reserved for failures in advance by an application, it can also be the task of psid to query the RMS for additional nodes to allow the application to proceed its execution. Whether a user has to allocate spare

nodes when submitting a job or if the RMS will have a pool of sparse nodes that can be used in case of a node failure, is an implementation detail we will cover in a later deliverable.[2]

5.3.3 Required changes in the ParaStation MPI communication library

In order to get a consistent checkpoint for a group of processes, and to not be affected by the *domino effect* [**ELNOZAHY02**], we use coordinated checkpointing. In such a schema, all processes of the group have to communicate to each other about the intent to take a snapshot. This is usually done using a two-phase commit protocol. In the case of an MPI application performing user-level checkpointing, the interface can be simplified though to a collective operation exhibiting barrier semantics. It will be implemented by the checkpoint/restart framework developed in the DEEP-ER project.

It is however not sufficient to coordinate all processes of a parallel application in order to ensure their local checkpoints form a valid global checkpoint suitable for restarting the application. It must also be guaranteed that no messages are “in-flight”, i.e. there must not be messages sent by one process but not received by any other process. This includes messages buffered somewhere within the MPI layer on either side. Such messages would not be sent again upon restart but would instead be lost, causing the application to malfunction.

Ensuring there are no “in-flight” messages while performing a checkpoint (which is a collective operation resembling barrier semantics in our model) is clearly the responsibility of the application. Nevertheless, to minimise debugging efforts, it might be useful for the application developer to get support from the ParaStation communication library detecting and reporting undelivered messages as an erroneous condition. We therefore plan to add a special version of a *barrier* which can be used by the checkpointing framework and implements this check for “in-flight” messages.

Additionally pscom needs to cope with network issues caused by node failures. For Infiniband, completion queues on nodes that communicated with a failed node can get in an undefined state and cause connections to unaffected nodes to also fail. The pscom library has to be extended to recover from such a situation without losing messages. We expect the Extoll interconnect to require similar treatment. Clean handling of such cases is especially important when parts of the parallel application should stay alive (see above).

5.3.4 Possible enhancements in the ParaStation management daemon

As noted earlier, the ParaStation management daemon is responsible for starting and monitoring a parallel application, as well as doing proper clean-up when the application ends. It is also responsible for all communication between the application and the RMS. Thus, it seems quite natural to also utilise the ParaStation management component for the communication between the application and the checkpoint/restart framework developed in the project. This includes support for evaluating and passing well-known environment variables on the one hand, and especially tailored API calls on the other hand. The details of this interaction will be defined in a later deliverable.

[2] Note that dynamic resource allocation is a topic covered in the upcoming deliverable D4.8 of the DEEP project. Developments for the DEEP-ER project will be based on the techniques implemented there.

SIONlib is a user-level library linked to the parallel application. In general it can only access files visible from at least one node involved in the execution of the parallel application. This is no issue when using the global file system to store data but might become of relevance when storing local checkpoints. In some cases (e.g. after restarting an application on a different set of nodes) it might be useful to a have an additional means of accessing non-local files. As the ParaStation daemons permanently run on every node in the system, and are connected via a scalable, reliable protocol, they could provide a framework for such a “remote file access”. Upon request, they could read and forward the contents of the corresponding file to the target node. However, as the daemons are usually set up to communicate via the management network, performance would not be optimal. We will therefore explore options to use the high-speed interconnect in such cases. In general, this would be an experimental feature to be implemented in close collaboration with the developers of SIONlib and the members of WP4.

6 DEEP-ER failure model

The goal of the failure model to be developed in the DEEP-ER project is twofold. First, we want to abstract the numerous possible component failures in the DEEP-ER Prototype in a way that matches the machine abstraction and the failure modes of Table 1, to be used when developing the CR mechanisms. Second, we want a model that allows us to estimate with reasonable accuracy the key parameters for tuning the CR mechanisms, such as frequency of checkpoints.

In this deliverable we present our main ideas for the failure model, which are the result of our early and on-going discussions. Thus, details of the model are still in flux and will be presented in later deliverables and documentation. Instead, we only present a high-level view of the model and highlight how it fits with the key ideas of the CR approach being developed as part of this work package.

For this purpose we propose to organise the failure model into two complementary categorisations. In the first level (machine model) we consolidate the various component failures in the DEEP-ER Prototype into coarser grain block failures (e.g., the entire NIC instead of its various sub-components). This categorisation can be used by the model to reason about the point on the CR hierarchy (see below) where the recovery should start. In the second (failure types, Table 1) we consolidate the various failure types as seen by the OS (e.g., a Corrected Error, regardless of where in the machine it happened). These two categorisations are then jointly used by the CR model to reason about the point on the CR hierarchy (see below) where the recovery should start and what CR approach to use.

With respect to the first level, in the machine model we consolidate individual component failures into the following types (we omit for now the Cluster Nodes as the focus of the resiliency model is currently on the Booster Nodes used for highly-scalable computation):

- Brick: encapsulates:
  - KNL: encapsulates:
    - CPU, main memory, etc.
  - NVM
  - NIC
- Brick network: encapsulates:
  - PCIe switch
  - Physical links
- NAM
- GPFS

We propose to model the CR approach hierarchically, with the following levels, which are specified with respect to the location of the checkpoint:

- Level 1: DRAM of buddy KNL card in same brick
- Level 2: NVMe in same brick
- Level 3: DRAM of buddy KNL card in remote brick
- Level 4: NAM
- Level 5: NVMe in remote brick
- Level 6: GPFS

This multilevel, hierarchical CR approach makes the following assumptions:

- Failures are independent
- Checkpoint intervals are fixed at each level
- Checkpoint requires stalling user application
- Monotonic increase in cost and resiliency at increased checkpoint level:
  - Cost of checkpointing at level k+1 is higher than at level k
  - Frequency of checkpointing at level k is higher than at level k+1
  - Checkpoint at level k+1 at time Ti subsumes all checkpoints at level k at time Tj prior to Ti
  - Cost of recovery from a checkpoint at level k+1 is higher than at level k
- Recovery happens with valid checkpoint at the lowest level possible

We propose to model the CR approach using a Markov model, similar to the approach used in [**ADAMMOODY10**]. Details of the model and how it is customised to meet the needs of the DEEP-ER hardware and software are still in development and will be discussed in a future report.

Reference

[**Frings; Wolf; Petkov (2009)**] Frings, W.; Wolf, F.; Petkov, V. (2009); Scalable Massively Parallel I/O to Task-Local Files; Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, Portland, Oregon, November 14 - 20, 2009, SC'09, SESSION: Technical papers, Article No. 17, New York, ACM, 2009. ISBN 978-1-60558-744-8. - S. 1 – 11 DOI: 10.1145/1654059.1654077

[**Freche; Frings; Sutmann (2010)**] Freche, J.; Frings, W.; Sutmann, G. (2010); High Throughput Parallel-I/O using SIONlib for Mesoscopic Particle Dynamics Simulations on Massively Parallel Computers Parallel Computing: From Multicores and GPU's to Petascale ed.: B. Chapman, F. Desprez, G.R. Joubert, A. Lichnewsky, F. Peters and T. Priol, Amsterdam, IOS Press, 2010. Advances in Parallel Computing Volume 19. - 978-1-60750-529-7. - S. 371 – 378 DOI: 10.3233/978-1-60750-530-3-371

[**KENTO12]** Kento Sato, Naoya Maruyama, Kathryn Mohror, Adam Moody, Todd Gamblin, Bronis R. de Supinski, and Satoshi Matsuoka. 2012. 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). IEEE Computer Society Press, Los Alamitos, CA, USA, , Article 19 , 10 pages.

[**ADAMMOODY10**] Adam Moody, Greg Bronevetsky, Kathryn Mohror, and Bronis R. de Supinski. 2010. 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). IEEE Computer Society, Washington, DC, USA, 1-11

[**FCAPELLO09**] Franck Cappello, Al Geist, Bill Gropp, Laxmikant Kale, Bill Kramer, and Marc Snir. 2009. Toward Exascale Resilience. *Int. J. High Perform. Comput. Appl.* 23, 4 (November 2009), 374-388.

[**BSCHROEDER09**] B. Schroeder, E. Pinheiro, W.-D. Weber. "DRAM errors in the wild: A Large-Scale Field Study." Sigmetrics/Performance 2009

[**EL-SAYED13**] El-Sayed, N.; Schroeder, B., "Reading between the lines of failure logs: Understanding how HPC systems fail," *Dependable Systems and Networks (DSN), 2013 43rd Annual IEEE/IFIP International Conference on* , vol., no., pp.1,12, 24-27. June 2013

[**AHWANG13**] A. Hwang, I. Stefanovici, B. Schroeder. "Cosmic rays don't strike twice: Understanding the characteristics of DRAM errors and the implications for system design." *Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2012)*.

[**INTELE7**] New RAS features of Intel Xeon e7 family. http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/xeon-e7-family-ras-server-paper.pdf

[**INTELMCA**] Enhanced Machine Check Architecture on Xeon processors. http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/enhanced-mca-logging-xeon-paper.pdf

[**ELNOZAHY02**] E. N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. 2002. A survey of rollback-recovery protocols in message-passing systems. ACM Comput. Surv. 34, 3 (September 2002), 375-408. DOI=10.1145/568522.568525 http://doi.acm.org/10.1145/568522.568525

[**FhGFS website**] The Fraunhofer Parallel File System: http://www.fhgfs.com

List of Acronyms and Abbreviations

A

API: Application Programming Interface

B

BADW-LRZ: Leibniz-Rechenzentrum der Bayerischen Akademie der Wissenschaften. Computing Centre, Garching, Germany

BN: Booster Node (functional entity)

BoP: Board of Partners for the DEEP-ER project

BSC: Barcelona Supercomputing Centre, Spain

C

CINECA: Consorzio Interuniversitario, Bologna, Italy

CN: Cluster Node (functional entity)

Coordinator: The contractual partner of the European Commission (EC) in the project

CPU: Central Processing Unit

D

DEEP: Dynamical Exascale Entry Platform

DEEP-ER: DEEP Extended Reach: this project

DEEP-ER Network: high performance network connecting the DEEP-ER BN, CN and NAM; to be selected off the shelf at the start of DEEP-ER

DEEP-ER Prototype: Demonstrator system for the extended DEEP Architecture, based on second generation Intel® Xeon PhiTM CPUs, connecting BN and CN via a single, uniform network and introducing NVM and NAM resources for parallel I/O and multi-level checkpointing

DEEP Architecture: Functional architecture of DEEP (e.g. concept of an integrated Cluster Booster Architecture), to be extended in the DEEP-ER project

DEEP System: The prototype machine based on the DEEP Architecture developed and installed by the DEEP project

DMA: Direct Memory Access

E

EC: European Commission

EU: European Union

Exaflop: 1018 Floating point operations per second

Exascale: Computer systems or Applications, which are able to run with a performance above 1018 Floating point operations per second

F

FhGFS: Fraunhofer Global File system, a high-performance parallel I/O system to be adapted to the extended DEEP Architecture and optimised for the DEEP-ER Prototype

FP7: European Commission 7th Framework Programme.

G

GRS: German Research School for Simulation Sciences GmbH, Aachen and Juelich, Germany

H

HPC: High Performance Computing

HW: Hardware

I

IB: InfiniBand

ICT: Information and Communication Technologies

IEEE: Institute of Electrical and Electronics Engineers

Intel: Intel Germany GmbH Feldkirchen,

IP: Intellectual Property

iPic3D: Programming code developed by the University of Leuven to simulate space weather

J

JUELICH: Forschungszentrum Jülich GmbH, Jülich, Germany

K

KNC: Knights Corner, Code name of a processor based on the MIC architecture. Its commercial name is Intel® Xeon PhiTM.

KNL: Knights Landing, second generation of Intel® Xeon PhiTM

KULeuven: Katholieke Universiteit Leuven, Belgium

L

M

MEW: Machine Evaluation Workshop

MIC: Intel Many Integrated Core architecture

Mont-Blanc: European scalable and power efficient HPC platform based on low-power embedded technology

MPI: Message Passing Interface, API specification typically used in parallel programs that allows processes to communicate with one another by sending and receiving messages

N

NAM: Network Attached Memory, nodes connected by the DEEP-ER network to the DEEP-ER BN and CN providing shared memory buffers/caches, one of the extensions to the DEEP Architecture proposed by DEEP-ER

NIC: Network Interface Card, Hardware component that connects a computer to a computer network

NVM: Non-Volatile Memory

O

OmpSs: BSC's Superscalar (Ss) for OpenMP

OpenMP: Open Multi-Processing, Application programming interface that support multiplatform shared memory multiprocessing

OS: Operating System

P

ParaStation MPI: Software for cluster management and control developed by ParTec

ParTec: ParTec Cluster Competence Center GmbH, Munich, Germany

PC: Personal Computer

PCI: Peripheral Component Interconnect, Computer bus for attaching hardware devices in a computer

PCIe: PCI Express, Standard for peripheral interconnect developed to replace the old standards PCI, improving their performance

PM: Person Month or Project Manager of the DEEP project (depending on the context)

PMT: Project Management Team of the DEEP-ER project

PR: Public Relations

PRACE: Partnership for Advanced Computing in Europe (EU project, European HPC infrastructure)

Q

R

S

SC: International Conference for High Performance Computing, Networking, Storage, and Analysis, organised in the USA by the Association for Computing Machinery (ACM) and the IEEE Computer Society

SISSA: International School of Advanced Studies, Trieste, Italy

SRA: Strategic Research Agenda prepared by ETP4HPC

SW: Software

T

TurboRVB: Quantum Monte Carlo Software for electronic structure calculations developed by SISSA

U

UREG: University of Regensburg, Germany

V

W

WP: Work Package

X

Y

Z