**NVM-ESR**: Using Non-Volatile Memory in Exact State Reconstruction of Preconditioned Conjugate Gradient

Yehonatan Fridman  
fridyeh@post.bgu.ac.il  
Ben-Gurion University of the Negev  
Israel Atomic Energy Commission  
Israel

Yaniv Snir  
yanivsn@post.bgu.ac.il  
Ben-Gurion University of the Negev  
Google  
Israel

Harel Levin  
harellevin@nrcn.org.il  
Scientific Computing Center, Nuclear Research Center – Negev  
Mobileye Vision Technologies  
Israel

Danny Hendler  
hendlerd@cs.bgu.ac.il  
Department of Computer Science,  
Ben-Gurion University of the Negev  
Israel

Hagit Attiya  
hagit@cs.technion.ac.il  
Department of Computer Science,  
Technion – Israel Institute of Technology  
Israel

Gal Oren*  
galoren@cs.technion.ac.il  
Department of Computer Science,  
Technion – Israel Institute of Technology  
Scientific Computing Center, Nuclear Research Center – Negev  
Israel

**ABSTRACT**

HPC systems are a critical resource for scientific research and advanced industries. The demand for computational power and memory is increasing and ushers in the exascale era, in which supercomputers are designed to provide enormous computing power to meet these needs. These complex supercomputers consist of many compute nodes and are consequently expected to experience frequent faults and crashes. Exact state reconstruction (ESR) has been proposed as a mechanism to alleviate the impact of frequent failures on long-term computations. ESR has shown great potential in the context of iterative linear algebra solvers, a key building block in numerous scientific applications.

Recent designs of supercomputers feature the emerging non-volatile memory (NVM) technology. For example, the Exascale Aurora supercomputer is planned to integrate Intel Optane™ DCPMM. This work investigates how NVM can be used to improve ESR so that it can scale to future exascale systems such as Aurora and provide enhanced resilience.

We propose the non-volatile memory ESR (NVM-ESR) mechanism. NVM-ESR demonstrates how NVM can be utilized in supercomputers for enabling efficient recovery from faults while requiring significantly smaller memory footprint and time overheads in comparison to ESR. We focus on the preconditioned conjugate gradient (PCG) iterative solver also studied in prior ESR research, because it is employed by the representative HPCG scientific benchmark.

The source code used by this work, as well as the benchmarks and other relevant sources, are available at: https://github.com/Scientific-Computing-Lab-NRCN/NVM-ESR.git.

**KEYWORDS**

NVRAM, Intel Optane DCPMM, MPI OSC, RDMA, PCG solver, ESR

*Corresponding author

1 INTRODUCTION

The past decade has seen a skyrocketing increase in the demand for high-performance computing (HPC) systems, in order to provide sufficient computing power for resource-hungry applications in various domains of science and economics. These needs ushered in a new exascale era of stronger and more complex supercomputers. With the increase in complexity and the number of compute nodes, comes an increased vulnerability to failures. Already for earlier generations of petascale computers, Schroeder and Gibson [20] have found that in certain situations, an application will be forced into recovery more than twice a day. This recovery frequency leads to a growing need for mechanisms to ensure efficient recovery from failures.

Exact state reconstruction (ESR) is a failure recovery mechanism, applicable to many distributed linear iterative solvers. Introduced by Chen [3], and refined by Pachajoa et al. [15], ESR provides fault tolerance by keeping redundancies of chosen process’ state variables in the memory of other processes. It piggybacks on the already existing transmission of data between processes that is required by the Sparse matrix–vector multiplication (SpMV) operation. In this manner, redundant copies of the input vector can be produced with relatively low memory and runtime overheads.

Necessary conditions on solvers for the applicability of ESR [14, 16, 17] are: (1) The iterative algorithm performs a finite-term recurrence, and (2) The iterative algorithm involves a matrix-vector product. In particular, ESR was applied to the Preconditioned Conjugate Gradient (PCG) solver, solving the linear equation $Ax = b$ for a symmetric positive definite matrix $A_{n \times n}$. Chen [3] introduced a way to use the sparse matrix-vector product to redundantly store
the input vector, as well as a way to reconstruct the entirety of the state of a preconditioned conjugate gradient (PCG) solver using the last two redundantly stored search directions and a scalar that was replicated in the cluster.

Redundancies in ESR are sent to multiple processes, in order to tolerate multiple simultaneous process failures. To maximize fault tolerance, processes replicate their state vectors at each of the other processes. This replication comes with a significant memory expense, as each process keeps state for each other process. Additionally, redundancies are sent all-to-all after each iteration of the computation (or after a certain period), leading to a surge in network traffic. To alleviate these problems, a modified ESR mechanism can replicate the state at only a fraction of the cluster, for example, at half of the processes. Nevertheless, even this mechanism comes with a significant memory overhead (see Section 3.1), leaving open the challenge of reducing the costs of ESR and improving its scalability.

Next-generation supercomputers, such as the flagship Aurora supercomputer [12], are expected to integrate non-volatile memory (NVM) devices, such as the emerging Intel Optane DC Persistent Memory Modules (DCPMM). When configured in memory mode, these devices are byte-addressable and can be used by processes as non-volatile random access memory (NVRAM). NVRAM’s ability to retain data even after node failures, opens new possibilities for high-performance computing, most notably, as a medium for data persistence upon node or process failure and consequent recovery. Previous investigation [8] of the ways in which Intel Optane DC Persistent Memory Modules can be used for supporting the recoverability of scientific applications has established that it enables enhanced checkpoint-restart support, either explicitly or transparently. This paper explores how NVRAM can be exploited to support ESR, while reducing the cost of redundancy, specifically in the context of PCG solvers. We present NVM-ESR, a new mechanism that utilizes NVRAM to improve the fault tolerance of linear solvers, while saving on the memory footprint. We study two possible architectures:

(1) Homogeneous NVRAM cluster, in which each compute node is equipped with its own NVM module, enabling the persistence of ESR state variables to local NVRAM by using either the persistent memory development kit (PMDK) [19] libraries or a local MPI window.

(2) NVRAM persistent recovery data (PRD) sub-cluster, in which recovery data is persisted in dedicated PRD sub-cluster nodes via remote MPI one-sided communication implemented using remote direct memory access (RDMA).

We note that, to the best of our knowledge, planned exascale supercomputers are not expected to use the homogeneous NVRAM architecture. This choice of architecture is due to the fact that integrating Optane DCPMM modules in each compute node would reduce the number of available DDR DIMM memory slots, thereby reducing the size of available DRAM, which is one of the crucial resources in HPC nodes. Hence, future supercomputers integrating Optane DCPMM modules are designed with remote NVM storage nodes, e.g., the DAOS storage server [16] in Aurora [12]. Nevertheless, we evaluate also the homogeneous NVRAM cluster NVM-ESR architecture, as future systems might include NVRAM in each compute node, for relatively cheap and fast byte-addressable memory expansion (see [8, Section 4]).

**Contribution**

In this work, we propose, implement and evaluate NVM-ESR, a novel NVRAM-based mechanism for scalable and resource-efficient exact state reconstruction for distributed linear iterative solvers. We consider both the homogeneous cluster architecture, in which each node stores recovery data locally, and a persistent recovery data (PRD) sub-cluster architecture, in which all recovery data is stored remotely on a sub-cluster of PRD nodes (in our system, we used a single PRD node). We implemented several variants of ESR, in which recovery data is stored in either DRAM, NVRAM, or an SSD storage device. We conducted a comprehensive performance evaluation of these implementations in terms of the memory and time overheads they incur.

The results of our evaluation show that NVM-ESR is significantly superior to ESR in terms of the size of the memory footprint required for storing the recovery data. Compared with NVM-ESR in the PRD sub-cluster architecture (henceforth denoted NVM-ESR/PRD), ESR incurs larger memory overhead by a factor proportional to the product of the total number of processes and the maximal number of simultaneous failures that the system can recover from. NVM-ESR is also superior to ESR in terms of the time overhead incurred by writing the recovery data, except for small numbers of processes that can fit inside a single node in the homogeneous architecture.

We implemented NVM-ESR/PRD by using MPI one-sided communication over InfiniBand’s remote direct memory access (RDMA) towards NVRAM and have optimized its usage by ESR’s persistence iterations (see Section 4.1). To the best of our knowledge, our work is the first to report on a scientific application implementation that accesses remote NVRAM in this manner. We show how to support the recoverability of such applications using NVRAM to substitute traditional checkpointing. The results of our experimental evaluation show that accessing remote NVRAM in this way is faster than accessing a remote SSD storage device in NVM-ESR/PRD, especially at high process counts. Moreover, in the homogeneous NVRAM architecture, the results show that accessing local NVRAM via a PMFS is faster than accessing it with MPI local windows or PMDK. (See Section 5.)

Based on the results of the empirical evaluation conducted in this work, we estimate that the small memory and time overheads of NVM-ESR/PRD will allow it, unlike ESR, to be deployed in future exascale supercomputers for providing high resiliency to the important class of solver algorithms for which ESR is suitable.

2 IN-MEMORY ESR AND ITS CHALLENGES

*Exact state reconstruction (ESR)* is a technique for recovering the state of a linear algebra iterative solver after a failure. ESR is applicable to iterative solvers that involve *sparse matrix-vector multiplication* (SpMV). The solvers perform a finite-term recurrence, hence the state can be reconstructed from a bounded amount of data. Furthermore, redundant copies of some vector variables can be produced with low memory and runtime overheads.
ESR first identifies the state of the solver. Upon recovery, the redundancy of vectors participating in the SpMV operations is used to reconstruct the other state vectors, by solving local equations on a replacement node. When the full state of the failed process is reconstructed, the computation can proceed on the replacement node. There is a generic strategy for identifying the state of an iterative linear algebra solver and for reconstructing the state upon recovery [14]. However, like prior work [16, 17], we focus on the preconditioned conjugate gradient (PCG) solver, which solves the linear equation $Ax=b$ for a symmetric positive definite matrix $A_{N\times N}$ (see Algorithm 1).

Algorithm 1 PCG solver for $Ax=b$.

1: $r(0) \leftarrow b - Ax(0)$, $z(0) \leftarrow p_{0}(0)$, $p(0) \leftarrow z(0)$
2: for $j = 0, 1, \ldots$ until convergence do
3: \quad $\alpha(j) \leftarrow r(j)^Tz(j)/r(j)^TAp(j)$
4: \quad $x(j+1) \leftarrow x(j) + \alpha(j)p(j)$
5: \quad $r(j+1) \leftarrow r(j) - \alpha(j)Ap(j)$
6: \quad $z(j+1) \leftarrow p_{j}(j+1)$
7: \quad $\beta(j) \leftarrow \frac{r(j+1)^Tz(j+1)}{r(j)^Tz(j)}$
8: \quad $p(j+1) \leftarrow z(j+1) + \beta(j)p(j)$
9: end for

In ESR, whenever the SpMV operation is applied, i.e., $Ap(j)$ is computed, the transition of $p$-values is augmented to create redundancy for all its entries. This augmented SpMV operation is denoted as $A_{spMV}$. We refer to the traditional ESR that creates redundancies on other processes’ RAM as in-memory ESR (see Algorithm 2). We refer to the set of all indices as $I$. The indices corresponding to a certain process $s$ are denoted by $I_s$. Specifically, $f$ denotes a failed process, and its indices are denoted $I_f$.

Algorithm 2 in-memory ESR for each iteration $j$ of two successive redundancy iterations of PCG. Line numbering refers to the lines of Algorithm 1.

3: \quad $a(j) \leftarrow r(j)^Tz(j)/r(j)^TA_{spMV}(A,p(j))$
4: \quad $\vdots$
8: \quad $p(j+1) \leftarrow z(j+1) + \beta(j)p(j)$

The ESR reconstruction phase for the PCG method on the replacement node appears in Algorithm 3. To reconstruct the complete PCG state of the failed process, two successive values of $p_{j}$ are required. Therefore, the redundancy for the $p$-values in the ESR model is saved for two successive iterations. In the reconstruction phase, the redundancy of $p_{j}$ is collected to the replacement node, and the values of $r_{j}$ and $x_{j}$ on the surviving nodes are collected as well. Solving some local linear equations, $r_{j}^{(1)}$ and $x_{j}^{(1)}$ are obtained on the replacement node. Together with the collected values of $r_{j}$, the whole state of the failed process $(r_{j}^{(1)}, x_{j}^{(1)}, p_{j})$ is reconstructed.

To tolerate multiple process failures, the redundancy should be saved in multiple copies. Thus, even when several processes crash together, values can still be recovered from the RAM of the surviving processes. If $c$ processes fail simultaneously, $c+1$ copies should be made. In this case, $I_f$ represents the indices of all the failed processes together, and the reconstruction algorithm is executed on several processes, solving the local equation systems together distributively.

There is a delicate balance between the runtime overhead required to save the state of the application securely, and the time it takes to recover after a failure.

ESRP [17] is a modification of ESR, where redundant copies are created every period to alleviate the networking overhead for each iteration. ESRP demonstrates a trade-off, where increasing the period of ESR decreases the runtime overhead, but increases the cost of discarding the iterations performed since the last storage stage was reached when recovery is required.

It is possible to minimize the amount of data being persisted, by a careful analysis of the application that identifies a minimum-sized set of variables whose state should be made persistent. These variables should be chosen such that all other significant variables can be reconstructed from their values. A generic method for this state identification for iterative solvers is described in [14]. It is also possible to take advantage of concurrent data distributed between nodes to reconstruct the state [16].

Finally, it is necessary to determine where the state is saved. In traditional systems, the state has to be saved in other cluster nodes [17], so that whenever a node fails, the surviving nodes send the state of the failed node to a spare node (Figure 1a).

NVM-based systems offer an attractive alternative solution to these challenges, as we describe next.

3 NVM-BASED ESR: OVERVIEW

To exploit NVRAM for ESR recovery, we use it for persisting the redundant data required for it instead of keeping it in the memory of other nodes. We name this mechanism NVM-ESR.

We do not deal here with the trade-off between the ESR period and the number of “wasted” iterations upon a failure demonstrated in ESRP [17] and simply assume that this period is chosen optimally according to the solver and the cluster characteristics. We focus instead on a single persistence iteration during the calculation. We
note that, as in ESRP, two successive iterations should persist their $p$-values in NVM-ESR.

In NVM-based systems, some processes $p_i$ have access to unique persistent memory space ($PM_i$), in addition to a unique volatile memory space ($VM_i$) available for every process. The total memory space of process $p_i$ is $TM_i = VM_i \cup PM_i$. Processes crash arbitrarily and independently of each other. The number of processes whose simultaneous failure is tolerated by the algorithm is denoted by $c$. Upon a crash of a process $p_i$, $TM_i$ becomes inaccessible and the contents of $VM_i$ are lost. When $p_i$ recovers, $PM_i$ becomes accessible again but $VM_i$ should be re-allocated and initialized.

There are two key possible ways to integrate NVRAM devices into a cluster architecture.

In the first, more traditional architecture, called homogeneous NVRAM cluster, each node has an NVRAM device attached to it (see Figure 1b). Each node’s state is saved into its own NVRAM whose coherency should be ensured by the application. In this model, if a process $p_i$ fails, $PM_i$ becomes inaccessible until $p_i$ recovers. In this architecture, $PM_i$ resides in the NVM of its node. If $p_i$ fails, the data can be recovered from the NVM once it recovers. For each failed process $p_i$, the reconstruction algorithm is executed on the same node that $p_i$ executed on before the failure.

In the second architecture there is sub-cluster of one or more persistent recovery data (PRD) nodes, each containing NVRAM modules, which stores recovery data for the rest of the cluster’s compute nodes (see Figure 1c). In this model, even if process $p_i$ fails, $PM_i$ remains accessible by all non-failed processes and nodes.

A failure of a PRD node renders its memory inaccessible, which may make it a single point of failure. This type of failure can be addressed by adding redundancy within the PRD sub-cluster itself, but this is outside the scope of this paper.

In the PRD sub-cluster architecture, recovery data is saved on the NVRAM sub-cluster, via the remote memory MPI’s One-sided API (to reduce overheads). (The details of how this is done are explained in Section 4.1.) When a process $f$ fails, its recovery data is accessible on the NVRAM sub-cluster and can be retrieved by a replacement node. For each of the failure processes, the reconstruction can be executed on any spare node that has access to the remote NVRAM Sub-Cluster.

The NVM-ESR persistence stage, as well as the NVM-ESR reconstruction phase for both these architectures, appear in Algorithm 4 and Algorithm 5, respectively.

### Algorithm 4 NVM-ESR for a persistence iteration $j$ of PCG. Line numbering refers to the lines of Algorithm 1.

```
3: $a^{(j)}$ ← $r^{(j)}Tz^{(j)}/r^{(j)}TAp^{(j)}$

8: $p^{(j+1)}$ ← $z^{(j+1)} + \beta^{(j)}p^{(j)}$

if homogeneous cluster then
  persist $p^{(j+1)}$ to $PM_f$ (local NVM)
if PRD sub-cluster then
  persist $p^{(j+1)}$ to remote NVM
```

### Algorithm 5 NVM-ESR Reconstruction phase of PCG. Line numbering refers to the lines of Algorithm 3.

```
if homogeneous cluster then
  Wait for failed nodes to recover and have access to $PM_f$
if PRD sub-cluster then
  Run reconstruction from any spare nodes that have access to the remote NVRAM Sub-Cluster

1: Retrieve the static data $A_{f,i}$, $P_{f,i}$ and $b_{f,i}$
2: $r_i^{(j)}$ and $x_i^{(j)}$

if homogeneous cluster then
  3: Read $\beta^{(j-1)}$, $p_{f,i}^{(j-1)}$ and $p_{i}^{(j)}$ from $PM_f$ (local NVM)
if PRD sub-cluster then
  3: Read $\beta^{(j-1)}$, $p_{f,i}^{(j-1)}$ and $p_{i}^{(j)}$ from remote NVM

8: Solve $A_{f,i}x_{i}^{(j)} = w$ for $x_{i}^{(j)}$
```

### 3.1 Comparing the Memory Utilization of in-memory ESR and NVM-ESR

Let $M_f$ denote the size of volatile memory in the system, $M_{NV}$ denote the size of non-volatile memory in the system, and $M = M_f + M_{NV}$ denote the overall size of memory in the system. Also, let $N$ denote the number of compute nodes, each with $t$ compute units. The number of the actual compute processes used by the solver is $\mathcal{P} = \mathcal{P}_f \cdot \mathcal{N}$. We consider a matrix $A$ of size $n \times n$. $M_{PCG}(n, \mathcal{P})$ denotes the amount of memory required by the PCG solver without any support for failure recovery. $M_{CG}(n, \mathcal{P}, \phi)$ denotes the amount of memory overhead required to support the recoverability of up to $\phi$ simultaneous process failures. In the following, we estimate $M_{PCG}(n, \mathcal{P})$ and $M_{CG}(n, \mathcal{P}, \phi)$ in the worst case of full fault tolerance (i.e., $\phi = \mathcal{P}$, and $\mathcal{P} = t \cdot N$), where $R$ is one of recoverability methods: in-memory ESR and NVM-ESR.

For the problem $Ax = b$, when $A$ is a sparse matrix of size $n \times n$, its representation at each node has $S \cdot n$ values, where $S$ is determined by the sparsity of the matrix. Each process also holds $n_{\mathcal{P}}$ values for each of $x$, $r$, $z$ and $p$. In the in-memory ESR model, in order to support full fault tolerance, $\mathcal{P} - 1$ redundancy copies of $p$ should be sent to all the other processes. Some of the values would be sent regardless as part of the SpMV operation to a few processes, but in the worst case, the amount of redundancy in ASpMV is $O(n)$ values for each process and hence $O(n \cdot \mathcal{P})$ values for all the processes together. All of these redundancies are transferred every persistence iteration via MPI communication between the processes and stored in RAM. We note that redundancies for $p^{(j-1)}$ should be available in addition to the redundancy values of $p^{(j)}$, so the total memory utilization for redundancies in the system is approximately $2 \cdot \mathcal{P} \cdot n$ values.

In the NVM-ESR model, however, no RAM is used for redundancy. In this model, for every value of $c$, only single copies of $p^{(j-1)}$ and $p^{(j)}$ are persisted to the NVRAM (which can be either local or remote, according to the NVRAM architecture as discussed earlier). If the NVRAM is local, the data is persisted directly from the CPU via
the memory bus, using the PMDK library [19] or local MPI windows over the NVRAM [4]. If NVRAM is remote, data is persisted via the remote memory MPI’s One-sided API [4] over RDMA. Thus, to ensure the resilience of the entire system, NVM-ESR does not store recovery data in RAM at all and stores only $O(n)$ recovery data values to the NVRAM, saving overall $O(\text{proc} \cdot n)$ space in $M$ relative to in-memory ESR.

Figure 2 depicts the memory usage of PCG for the 7-point stencil of a 3-D Poisson equation executed on the cluster described in Figure 6. For each execution, the problem size changes, such that RAM utilization is fixed for each process in such a way that when a node is fully occupied (with 32 processes), its entire RAM is used. The figure shows the size of problems that can be handled with in-memory ESR decreases because recovery data occupy a significant part of the node DRAM.

Example: The Argonne National Laboratory’s next-generation flagship supercomputer, Aurora [21], is planned to be one of the world’s first exascale systems. It is planned to consist of 9000 compute nodes, each with 112 CPU cores ($\sim 10^6$ cores total). Total system memory is estimated at 10PB. In addition, it is expected to have over 230PB of high-performance storage, including Intel Optane™ DC SSDs and DCPMMs (with DAOS). For a PCG 7-point stencil of a 3-D Poisson equation, extrapolating the in-memory ESR PCG RAM consumption presented in Figure 2 into Aurora scale, we estimate in-memory full fault tolerance ESR RAM consumption to be $\sim 30\%$ of the system memory, hence $\geq 3$PB. NVM-ESR suggests eliminating this memory overhead at the expense of only 3PB of NVRAM because every value that resides in the RAM of $\sim 10^6$ nodes can now be persisted to NVRAM only once. Even if one keeps just a single copy of redundancy at each compute node, the saving in RAM usage remains significant: In such a case, only $\sim 10^6 \div 112 = 9000$ copies are made, instead of occupying $3$PB of RAM in-memory as would be done by ESR for the previous problem. Larger problems can be executed when fewer redundancies are kept and each copy is larger. Hence, savings in RAM when the problem fully occupies Aurora would be significantly larger than 26.7TB.

### 3.2 NVM-ESR vs. Checkpoint Restart

In transparent checkpoint restart (for example with DMTCP [2]), the entire memory image of the program is persisted in a storage device every certain number of iterations or a certain period. Because the memory footprint of the PCG application includes the full problem state as well as the static data of the solver, the overhead for persisting the full image of the application is dramatically increased (with additional overheads of DMTCP itself). ESR inherently alleviates this overhead by persisting only a subset of the

| Nodes            | 9,000         |
|------------------|---------------|
| Sockets          | 2 (per node)  |
| CPU Spec. Scalable | 56 cores X Intel Xeon “Sapphire Rapids” processor (per socket) |
| Aggregate System Memory | $\geq 10PB$ |
| High-Performance Storage | $\geq 230PB$, including Intel Optane™ DC SSDs and DCPMMs |
| Network switch   | 64-port switch, 25GB/s per direction |
| Peak Performance | 2 exaFLOPS (expected speed) |

Figure 3: Aurora Supercomputer expected relevant specifications.
state and computationally reconstructing the rest of the state when recovering after a failure. In addition, the NVM-ESR mechanism persists the recovery data to the NVRAM directly, either locally via the memory bus or remotely via RDMA. Unlike traditional C/R systems, this eliminates the need to have the application access a traditional storage device by invoking I/O operations in block granularity.

4 THE IMPLEMENTATION IN DETAIL

This section describes the various ways in which NVM-ESR can use the system’s capabilities to persist data to NVRAM.

4.1 MPI One-Sided Communication (OSC) over RDMA

Remote direct memory access (RDMA) support in networks became mainstream, particularly through the widespread adoption of InfiniBand as a commodity network fabric [9]. MPI-1 provides a powerful and complete interface for the message-passing approach. MPI-2 added (and MPI-3 greatly extended) remote memory operations that provides a way to access the memory of remote processes directly, through operations that put data to, get data from, or update data at a remote process. Unlike message passing (using standard send and receive operations), the program running on the remote process does not need to call any routines to match the put or get operations. Thus, remote memory operations can provide better performance for distributed and parallel programs. This functionality of remote memory access is implemented via memory windows. The term window is used since MPI limits what part of a process’s address space is accessible to other processes.

Several works (e.g., [4, 6, 13]) present different persistence schemes to correctly utilize RDMA over NVRAM. Dorozynski et al. [4] incorporate non-volatile RAM into wrappers over MPI One-Sided API, in order to provide persistence of data stored in MPI windows. They extend the RDMA MPI One-Sided Communication (OSC) to persist the data stored in windows to NVRAM. Data consistency is ensured by copying necessary data into a separate location. More specifically, the data is saved into two locations alternately in order to have at least one proper “checkpoint” even if a failure occurs during checkpointing, creation. An implementation of this method [5] provides a new programming model by allowing processes to communicate freely using standard OSC functions and fall back to a state saved during synchronization. Moreover, it supports various synchronization calls of OSC, as we will explain next.

RDMA communication calls of the MPI ISC API must occur in the invoking process only within an access epoch for the window. The transferred data is available only when exiting the access epoch. Such an epoch starts with an RDMA synchronization call on the window; it then proceeds with zero or more RDMA communication calls (e.g., MPI_PUT or MPI_GET) on the window; it completes with another synchronization call on the window [7].

RDMA communications fall in two categories: active target communication and passive target communication.

In active target communication, the data is moved from one process to another, and both processes are explicitly involved in the communication. In contrast to standard message passing, in active target communication RDMA operations are controlled only by the invoking process, and the target process only participates in the synchronization. In active target communication, a target window can be accessed by RDMA operations only within an exposure epoch. Distinct exposure epochs at a process on the same window must be disjoint, but such an exposure epoch may overlap with multiple access epochs for the same window.

MPI provides two synchronization mechanisms for active target communication:

1. A general collective RMA synchronization, in which an access epoch at an origin process or an exposure epoch at a target process are started and completed by calls to MPI_Win_Fence.
2. Synchronization in which only pairs of communicating processes synchronize using the Post-Start-Complete-Wait (PSCW) protocol. In PSCW, an access epoch is started at the origin process by a call to MPI_Win_Start and is terminated by a call to MPI_Win_Complete. An exposure epoch is started at the target process by a call to MPI_Win_Post and is completed by a call to MPI_Win_Wait. The post-call has a group argument that specifies the set of origin processes for that epoch.

In passive target communication, the target process does not execute RMA synchronization calls, and there is no notion of an exposure epoch. Instead, passive target synchronization is accomplished by using MPI locks at the origin process with MPI_Win_Lock and MPI_Win_Unlock. It is used to ensure that data operations from other processes do not modify the data unexpectedly.

Dorozynski et al. [5] support all these communication mechanisms in their extension of MPI OSC over NVRAM. For the active target synchronization, exposure epochs are closed with MPI_Win_Fence_persist (for active target communication) or MPI_Win_Wait_persist (for passive target communication), to ensure that data reaches NVRAM before exiting the exposure epoch. To implement NVM-ESR in the NVRAM PRD sub-cluster architecture, it is necessary to ensure that data is persisted successfully to the NVRAM in the PRD node after each persistence iteration and before the successive persistence iteration attempts to access the window. Consequently, the target process must know when the exposure epoch is closed, and thus, an active target mechanism is more suitable for NVM-ESR.

Since a persistence iteration usually requires a significant period of time, we can optimize by releasing the access epochs of the compute processes while the target process is still persisting the data in its exposure epoch. This allows compute processes to proceed to the next compute iteration. For this reason, we choose the PSCW mechanism to be applied in NVM-ESR. Within the access epoch, a process executes a MPI_Win_Put_pmem to transmit data to the remote process, and MPI_Win_Get_pmem to read the data when recovery is needed. Whenever a compute process completes its data operations with MPI_Win_Complete, it exits the access epoch and proceeds. Figure 4 illustrates a PSCW epoch for the MPI OSC to an NVRAM PRD node in a persistence iteration.

A similar implementation for the homogeneous NVRAM cluster architecture, which we also consider, uses a separate local window for each process. In this architecture, each process accesses its own local window and persists its data locally.
4.2 Persistent Memory Development Kit (PMDK)

PMDK [19] offers a set of user-space APIs and interfaces to interact with non-volatile memory, with support for multiple abstraction layers, over Linux and Windows. PMDK libraries are designed to leverage the direct access allowed by persistent memory as much as possible. Persistent libraries in PMDK help applications maintain data structure consistency in the presence of failures. One such library is \textit{libpmemobj}, which helps the programmer manage persistent memory arrays and data structures. We implement NVM-ESR using the \textit{libpmemobj} library to persist ESR data directly to local NVRAM in the homogeneous NVRAM cluster architecture. Each process first creates a persistent memory pool using a call to \textit{pmemobj_create} and then, at each persistence iteration, persists ESR data using \textit{pmemobj_persist}. Figure 5 illustrates a persistence iteration using \textit{libpmemobj} in the homogeneous NVRAM cluster architecture.

PMDK also provides \textit{librpmem}, a remote RDMA access library, which supports remote access to persistent memory, with a synchronous write model. The local initiator writes and all of the remotely replicated writes must complete before the local write is completed back to the application. This library can be useful in the implementation of NVM-ESR in the PRD architecture, although this is out of the scope of this paper.

4.3 Persistent Memory File Systems (PMFSs)

For comparison, we also evaluate the option of persisting data to NVRAM using Persistent Memory File Systems (PMFSs). PMFSs can be either local or distributed. These file systems often exploit the byte-addressability of NVRAM. They support a special mode called Direct Access (DAX) that enables memory mapping directly from the NVRAM to the application memory space. This mapping bypasses the kernel, page cache, and I/O subsystem, avoids interrupts and context switching, and allows the application to perform byte-addressable load/store memory operations [1].

Fridman et al. [8] present an evaluation of local PMFSs (e.g., ext4-DAX and SplitFS [11]) for writing and reading diagnostics of scientific applications, as well as for performing C/R of these applications using transparent or explicit checkpointing. While these local PMFSs are good choices for single-node workloads, scientific computing applications typically require a distributed PMFS operating on multiple nodes. NVM-ESR can utilize local PMFSs in the homogeneous NVRAM cluster architecture to persist the recovery data of each process locally in its node. Distributed PMFSs, however, control how data is stored and retrieved from NVRAM when accessed by more than one node, using network communication. Distributed PMFSs can be utilized by NVM-ESR in the PRD sub-cluster architecture to persist recovery data remotely on the NVRAM present on PRD nodes.

In this work, we implement NVM-ESR for the homogeneous NVRAM cluster architecture by using ext4-DAX as a local PMFS over the local NVRAM.

5 EVALUATION

To evaluate NVM-ESR’s performance, we employed it for the PCG solver and compared it with the fully fault-tolerant ESR. Our experimental cluster specifications are listed in Figure 6. Our cluster consists of 8 compute nodes (each with 32 compute cores and 128GB DRAM) and a single NVRAM node (with 20 compute cores, 192GB DRAM, and 1TB Intel Optane™ DCPMM, populated as in Figure 7).

We evaluated both the homogeneous NVRAM cluster and the NVRAM PRD sub-cluster architectures. As our cluster contains only a single NVRAM node, our evaluation of the homogeneous NVRAM cluster architecture is limited to 20 processes. Nevertheless, this evaluation can be reliably extrapolated to a multi-node homogeneous NVRAM cluster architecture, since persisting data from each process to its local node is an embarrassingly parallel workload at the node level.

For the evaluation of NVM-ESR/PRD, we use the NVRAM node in our experimental cluster as a single PRD node. Multiple NVRAM nodes can serve as the PRD sub-cluster, distribute the persistent data to eliminate bandwidth bottlenecks or create a RAID over the sub-cluster to increase fault tolerance.

Figure 8 shows an estimate of NVRAM utilization of NVM-ESR on our cluster. The graph on the left shows the amount of NVRAM required for different numbers of processes when a fixed amount
We next present the time overheads of a single persistence iteration in NVM-ESR PCG for a 7-point stencil of a 3-D Poisson equation (with a fixed size for local vectors of 176,400 entries each). The time overhead of a single redundancy iteration of in-memory ESR with full fault tolerance are presented as well. We do not show the time overheads for the reconstruction phase because the number of reconstruction phases throughout the execution can be assumed to be much smaller than that of persistence iterations. Moreover, the reconstruction phase is heavily governed by the full-state reconstruction calculations, as explained in [3].

Figure 9 presents the time overheads of ESR and NVM-ESR in the homogeneous cluster architecture. We have implemented persistence to the local NVRAM using the ext4-dax local PMFS, PMDK, and MPI local windows over NVRAM. For reference, we also measured the time overhead of persisting the ESR data to a local SATA-SSD device. Dashed lines refer to a natural extrapolation of the results beyond a single NVRAM node\(^1\), based on the simple observation that local persistence operations in different nodes proceed in parallel. Above 32 processes, ESR’s time overhead increases significantly since persistence data must be sent to the RAM of processes in remote nodes.

Figure 10 presents the evaluation of a single persistence iteration of NVM-ESR in the PRD sub-cluster architecture implemented with MPI OSC over RDMA to NVRAM. To show the cost of implementing MPI OSC over NVRAM, we also present the time overhead of MPI OSC over RDMA when the windows are on RAM without persistent operations. For reference, we measure the time overhead of persisting the ESR data to a remote SATA-SSD device via SSH-FS. It can be seen that the overhead of ensuring persistence is relatively small. It can also be seen that MPI-OSC over RDMA access to NVRAM is faster than accessing a remote SSD storage device, especially at high process counts.

These results show that NVM-ESR/PRD is significantly superior to ESR in terms of the time overhead incurred by writing the recovery data, except when the number of processes is small, and they all fit inside a single node.

6 DISCUSSION

This work presents NVM-ESR, a mechanism for utilizing NVRAM to increase the resiliency of exact state reconstruction. Our extensive evaluation shows that NVM-ESR supports instant recovery while decreasing the memory footprint and time overheads. We focus on the preconditioned conjugate gradient iterative solver, but our results are readily applicable to other ESR-enabled iterative solvers. A natural challenge is to extend NVM-ESR (and ESR, in general) to non-iterative solvers.

\(^1\)We remind the reader that our single NVRAM node contains 20 cores.
Figure 9: Time overhead (in log-scale) for NVM-ESR/ESR for single persistence/redundancy iteration in crash-free CG computation in the homogeneous cluster architecture.

Figure 10: Time overhead for NVM-ESR/ESR for single persistence/redundancy iteration in crash-free CG computation in the PRD sub-cluster architecture. The top chart shows the general trend, while the lower chart zooms in (in log scale) on the behavior with $\leq 32$ processes.
Going beyond recovery, the techniques we developed for using one-sided MPI windows open new avenues for using direct memory access to NVRAM. For example, as an effective storage extension for Exascale HPC systems, in particular, in future deployment of DAOS [10] on Aurora.

We also intend to evaluate the performance, under various access patterns, of remote storage based on SSD, MVME-SSD, and NVRAM.

ACKNOWLEDGMENTS

This work was supported by Pazy grant 226/20 and the Lynn and William Frankel Center for Computer Science. Computational support was provided by the NegevHPC project [18]. The authors would like to thank Gabi Dadush, Israel Hen and Emil Malka for their hardware support.

REFERENCES

[1] Intel. Quick Start Guide Part 1: Persistent Memory Provisioning Introduc- tion. https://software.intel.com/content/www/us/en/develop/articles/qsg-intro-
to-provisioning-pmem.html.
[2] Jason Ansel, Kapil Arya, and Gene Cooperman. 2009. DMTCP: Transparent checkpointing for cluster computations and the desktop. In 2009 IEEE International Symposium on Parallel & Distributed Processing, IEEE, 1–12.
[3] Zizhong Chen. 2011. Algorithm-based recovery for iterative methods without checkpointing. In Proceedings of the 20th International Symposium on High Performance Parallel and Distributed Computing. ACM, 73–84. https://doi.org/10.1145/1966130.1966142
[4] Piotr Dorożyński and Paweł Paweł. 2016. Extension of MPI One-Sided Communication API. https://github.com/pmem/pmem-ext/raw/master/mpi_one_sided_extension/doc/mpi_one_sided_extension.pdf. https://github.com/pmem/pmem-ext/raw/master/mpi_one_sided_extension/doc/mpi_one_sided_extension.pdf
[5] Jingwen Du, Fang Wang, Dan Feng, Wenguang Li, and Fan Li. 2021. Fast and Consistent Remote Direct Access to Non-volatile Memory. In 50th International Conference on Parallel Processing 1–11.
[6] Donald E. Knuth. 1997. The art of computer programming, volume 3: Sorting and searching. Addison-Wesley.
[7] Message Passing Interface Forum. 2015. MPI: a standard message passing interface. https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf.
[8] Yehonatan Fridman, Yaniv Snir, Matan Rusovoykovsky, Kfir Zvi, Harel Levin, Danny Hendler, Hagit Attiya, and Gal Oren. 2021. Assessing the Use Cases of Persistent Memory in High-Performance Scientific Computing. In 2021 IEEE/ACM 11th Workshop on Fault Tolerance for HPC at Xtreme Scale (FTXS). IEEE, 11–20.
[9] Bill Gropp, Tony Hoefler, Rajeev Thakur, and Ewing Lusk. 2009. Using Advanced MPI. Massachusetts Institute of Technology.
[10] Michael Hennecke. 2020. Daos: A scale-out high performance storage stack for storage class memory. Supercomputing frontiers. 40.
[11] Mohammad Kadhedi, Se Kwon Lee, Sanidhya Kashyap, Taeseo Kim, Ashleesh Kolli, and Vijay Chidambaram. 2019. SplitFS: Reducing software overhead in file systems for persistent memory. In Proceedings of the 27th ACM Symposium on Operating Systems Principles. 494–508.
[12] Argonne National Laboratory. 2022. Aurora. https://www.alcf.anl.gov/aurora/.
[13] Xinhun Liu, Yu Hua, Xuan Li, and Qifan Liu. 2019. Write-optimized and consistent RDMA-based NVM systems. arXiv preprint arXiv:1906.08173 (2019).
[14] Carlos Pachajna, Robert Ernstbrunner, and Wilfried N Gansterer. 2020. A Generic Strategy for Node-Failure Resilience for Certain Iterative Linear Algebra Methods. In 2020 IEEE/ACM 10th Workshop on Fault Tolerance for HPC at eXtreme Scale (FTXS). IEEE, 41–50.
[15] Carlos Pachajna, Markus Levonyak, and Wilfried N Gansterer. 2018. Extending and Evaluating Fault-Tolerant Preconditioned Conjugate Gradient Methods. In 2018 IEEE/ACM Workshop on Fault Tolerance for HPC at eXtreme Scale (FTXS). IEEE, 49–58.
[16] Carlos Pachajna, Markus Levonyak, Wilfried N Gansterer, and Jesper Larsson. Truff. 2019. How to make the preconditioned conjugate gradient method resilient against multiple node failures. In Proceedings of the 48th international conference on parallel processing. 1–10.
[17] Carlos Pachajna, Christina Pacher, Markus Levonyak, and Wilfried N Gansterer. 2020. Algorithm-based checkpoint-recovery for the conjugate gradient method. In 49th international conference on parallel processing-ipp. 1–11.