Exploiting Global View for Resilience (GVR)

Andrew A. Chien (PI), Hajime Fujita, Zachary Rubenstein, Aiman Fang, Nan Dun, University of Chicago; Pavan Balaji (co-PI), Kamil Iskra, Pete Beckman, Jeff Hammond, Wesley Bland, ANL

http://gvr.cs.uchicago.edu/

Background
As ever system scale increases and process shrinks, it is anticipated that future large-scale machines will experience higher error rates. Of particular concern among types of errors are soft errors, which can cause latent errors (often called silent data corruption). Soft errors will be one of the most crucial issues in exascale systems. In order to utilize systems that are unreliable and subject to complex errors, a flexible and powerful programming model and tool that allow flexible recovery are needed.

Big Ideas
We are developing the Global View Resilience (GVR) library to make large scale applications reliable on unreliable hardware. The key ideas of GVR are the following:

- **Multi-version, multi-stream distributed arrays**: GVR provides globally-visible, distributed arrays as a foundation of reliable execution [8, 5]. Applications store critical data to these arrays, and utilize them for recovery in case of errors. A GVR array can be multi-versioned, which means it can preserve multiple snapshots of its contents, and an application can access any version at any time. The application declares an opportunity to take a version when it knows that the content of the array is consistent across multiple users (processes) of the array. An application can have multiple arrays at the same time (multi-stream) and can choose different snapshot frequencies for different arrays. Multi-versioning enables recovery from latent errors, which cannot be covered by traditional checkpoint/restart scheme. Multi-stream allows applications to apply a different recovery strategy for each array, leading to flexible, efficient recovery.

- **Open resilience**: in GVR, the fundamental philosophy about error recovery is that application programmer can potentially control anything. Giving the application programmer control allows computer systems to shift from the traditional “fail-stop” model to an “error signaling & handling” model. This shift maximizes the chance for an application to recover from errors. Therefore the entire computer system, from hardware to applications, must be reorganized so that it allows applications to handle various kinds of errors. GVR works as a mediator between applications and the rest of the computer system by providing a unified error signaling & handling interface. Using GVR to handle errors maximizes the recoverable errors, as well as outcome of investment for resilience.

Incremental Investments to Resilience
The GVR library is a user-level library, so it does not require any special compiler or any other kinds of tools. GVR also does not require any architectural change to existing applications. While GVR provides PGAS-style distributed arrays, GVR does not require that the entire program be rewritten in that style. Application programmers can apply GVR only to the part of the data structure that needs to be resilient. For these reasons, it is quite easy to apply GVR to existing applications. As a consequence, required code change is small. This allows existing applications to take a gentle slope towards running on large-scale, unreliable platforms.
Current Accomplishments

- Working implementation of the GVR library, with basic array manipulation APIs and in-memory versioning, as well as error handling and signaling APIs[1]
- Formal modeling of multi-version snapshots to reveal when and how multi-versioning is useful in an environment that admits latent errors
- Log-structured implementation of distributed arrays for preserving multiple versions efficiently
- Various application and library studies[2]

Application/Library Studies

We prove that GVR can be easily applied to many real scientific applications by augmenting existing applications with GVR in many different ways. Many of these studies are also outcomes of collaboration with other DoE national labs and co-design centers.

- ddcMD, production-level molecular dynamics code, a collaboration with LLNL[3]
- OpenMC, production-level nuclear reactor simulation with Monte Carlo method, a collaboration with CESAR co-design center[6]
- Trilinos, library for scientific computing, a collaboration with SNL[7]
- Linear solvers using Trilinos, a collaboration with SNL
- Chombo, adaptive mesh refinement framework, a collaboration with ExReDi project at LBL

Impacts

- Enables "end-to-end", incremental resilience investment for large-scale applications as error rate increases
- Enables efficient, portable control of resilience coverage and overhead
- Facilitates HW vendors, OS vendors, runtime developers, as well as application developers to collaborate towards more effective resilience

Future Work

- Public preview release
- Broader application and library work
- Engagement in X-Stack programming models
- Grow Open Resilience engagement with OSR, X-stack runtime, programming model, Fast Forward architecture teams
- Deeper research on library implementation, especially on exploiting opportunities in novel storage and memory hierarchies

References

3. Aiman Fang and Andrew A. Chien, Applying GVR to Molecular Dynamics: Enabling Resilience for Scientific Computations, TR-2014-04, CS Department, Univ. of Chicago
Efficient Array Versioning for Global View Resilience

Hajime Fujita, Nan Dun, Zachary Rubenstein, Andrew A. Chien, University of Chicago; Kamil Iskra, Pavan Balaji, ANL
http://gvr.cs.uchicago.edu/

Background
Global View Resilience (GVR) provides a multi-version distributed array, which allows applications to preserve multiple snapshots of the array contents. Having multiple versions allows application programmers to recover from complex errors, such as latent errors. However, how to implement efficient versioning remains as a critical question. We are exploring two approaches for efficient versioning, based on an observation that many applications modify only a limited portion of the array for each version.

Approach 1: Log-structured Array

The first approach is to introduce a special in-memory data structure called log-structured array for multi-version array, which is inspired by log-structured file system [1]. A memory buffer forms a log, and updates to the array are appended to the tail of the log. Each version has a set of metadata blocks, which work as indices to actual data blocks. So the array access incurs two steps, first metadata access and then access to a data block pointed by the metadata. In this way, since versions are incrementally created on each write, versioning operation completes very fast. We designed the log-structured array so that fundamental data access (e.g., put/get/acc) can be performed only using one-sided remote memory operations. For example we use fixed-size data blocks in order to simplify metadata access.

We applied the log-structured array to three different applications (OpenMC, PCG solver, canneal from the PARSEC benchmark suite), and compared the performance and memory usage against flat, contiguous array implementation called “flat array”, where versioning is done by copying the entire array. Figure 2(left) shows that versioning runtime overheads can be negligible (3.7% for PCG, 4.7% for OpenMC), and manageable for the other (26% for canneal). Figure 2(right) shows the relative memory consumption of each application, including the maximum memory savings available when the block size of the log-structured array is set equal to the message size. The result shows that the log-structured array saves as much as 97.7% memory for preserving versions for canneal. Full results for the log-structured array study can be found in [2].

Approach 2: Study on Change Tracking Techniques

The second approach is to track changes on a memory buffer between versions. We designed three approaches for managing change tracking.
1. **User:** User-supplied dirty-bit tracking, in which a user of the versioning system specifies which area is modified. In GVR, most of the array operations are done via explicit function calls such as *put* or *acc*, so the GVR library can internally generate dirty bit information. This allows fine-grain dirty bit tracking.

2. **Kernel:** Kernel-level page-based memory tracking, which utilizes page protection interface (e.g. *mprotect*) provided by operating system kernel. All the pages in the buffer are write-protected at the beginning of the version, and then the GVR library catches access violation signal (*SIGSEGV*), records dirty page, allows write access to the page, then resume execution. Dirty bit tracking granularity is page size.

3. **Hardware:** Hardware-accelerated dirty-bit tracking, which utilizes dirty page bit tracking feature in CPU. This method is faster than the kernel-level page-based tracking because it does not require page fault and signal handling, but it requires special enhancements to an operating system kernel.

![Figure 3: Performance comparisons among change tracking approaches](image)

We compared the performance of these three approaches using random walk on 128MB buffer, based on RandomAccess kernel from the HPC Challenge benchmark suite. Figure 3 shows that “User” approach is the best for frequent snapshots, “Kernel” has a large overhead and thus “Hardware” should be used instead of “Kernel” where OS kernel modification is allowed.

**Future Work**
- Studying compression, differencing for further reducing version size, adding redundancy for protecting data
- Comprehensive application-level evaluation of change tracking approaches

**Acknowledgements**
We gratefully acknowledge the computing resources provided on Midway, high-performance computing cluster operated by the Research Computing Center at The University of Chicago.

**References**
Flexible Rollback and Forward Error Recovery in AMR Using Global View Resilience

Zachary Rubenstein, Hajime Fujita, Andrew A. Chien, University of Chicago; Anshu Dubey, Brian van Straalen, Lawrence Berkeley National Laboratory
http://gvr.cs.uchicago.edu/

Background
Adaptive mesh refinement (AMR) is a critical numerical technique for several important HPC scientific applications. Chombo[1, 2] is a software framework for block-structured AMR. We applied the Global View Resilience (GVR) library to Chombo in order to allow more flexible error recovery using rollback and forward recovery. Our GVR exploits the application-level state capture infrastructure which Chombo already has.

Approach
Chombo utilizes two key features of GVR.
- **Multi-version, multi-stream distributed arrays:** the GVR library provides distributed arrays that can preserve multiple versions. The AMR hierarchy has levels, and each level has its own time-step, so Chombo maps one GVR array to one level. Each level is versioned separately, and does its own recovery. By combining GVR and the existing state capturing mechanism in Chombo, we could easily recover from various data corruption errors.
- **Open resilience:** the GVR library works as a central hub for error signaling and handling. GVR provides a unified interface for error signaling and handling. This allows Chombo to generalize its error handler for data corruption to broader classes of errors such as process/node failure. We exploit ULFM, a library that tolerates low-level communication failure such as process crashes or network failure. Upon failure, ULFM signals an error to GVR, then GVR signals Chombo. Because of the unified error handling interface in GVR, Chombo can reuse the data corruption error handler for process/node failures as well. This indicates that GVR unified error handling interface maximizes recoverable errors, encouraging application programmers to invest in writing rich error handlers.

Future Perspectives
GVR enables several different recovery schemes:
- Localized recovery that exploits the multi-version, multi-stream arrays provided by GVR, i.e. run the recovery procedure only for affected data structure or processes
- Forward recovery that exploits AMR-specific application domain knowledge [5]
- Real GVR implementation that utilizes ULFM and tolerates real process failures

Figure 1: Architecture of GVR-augmented Chombo
Current Accomplishments
We developed a prototype of GVR-augmented Chombo, to prove that our approach is promising.

- Enhanced GVR’s unified error handling interface so that the application can handle a process failure event with GVR
- GVR simulates process failure and signals the failure event to the application
- Chombo receives the signal and rebalances and reloads the level data so that it can continue computation with a smaller number of processes
- About 300 lines of new code were added, and 400 lines of code were repurposed, in Chombo, which has about 547K lines in total.

Impacts
- Proves the benefit of open resilience, encouraging scientific computing community and vendors to engage in open resilience
- Synergy with ExReDi project at LBNL
- Same idea can be applied to BoxLib, a selected application by the ExaCT co-design center

Acknowledgement
We acknowledge our collaborator at Argonne National Laboratory: Wesley Bland. This work is a collaboration between the Global View Resilience project and the ExReDi project under the RXSolver project.

References
- http://fault-tolerance.org/

main () {
/* Allocate GVR distributed arrays for preserving level header and data */
GDS_alloc(&gds_h);
foreach (level) GDS_alloc(&gds_level[level]);
/* Associate an error handler with the level header array */
GDS_register_global_error_handler(gds_h,
GDS_ERROR_PROC_FAILURE, error_handler);
loop {
   do_computation();
   /* Dump current level data to GVR arrays */
   save_state(gds_h, gds_level);
   /* Save snapshots of each array */
   GDS_version_inc(gds_h);
   foreach (level) GDS_version_inc(gds_level[level]);
} /* Inject failure - simulate one process crash */
if (procID == failedProc)
   GDS_simulate_proc_failure(gds_h);
GDS_fence(gds);
/* Reconstruct the data distribution to adapt to smaller number of processes */
rebalance(n_procs);
/* Restore level data from GVR arrays */
restore_state(gds_h, gds_level);
/* Resume computation */
loop {
   do_computation();
}
/* Error handler - invoked upon process failure event */
status_t error_handler(gds, error_desc) {
   /* Obtain a new communicator that reflects the smaller number of processes */
   get_comms(error_desc, &old_comm, &new_comm);
   Chombo::comm = new_comm;
   GDS_resume_global(gds, error_desc);
}

Figure 2: Pseudo-code of GVR-augmented Chombo
Applying GVR to Molecular Dynamics: Enabling Resilience for Scientific Computations

Aiman Fang, Andrew A. Chien, University of Chicago; http://gvr.cs.uchicago.edu/

Background

Scientific and engineering computations have driven the demand for large-scale computing. Molecular dynamics codes, which simulate the movements of a collection of atoms/particles, are an important computational method in a wide variety of areas of biology, chemistry, and physics. We applied the GVR (global view resilience) library to the ddcMD (domain decomposition molecular dynamics) code, developed by LLNL, both to explore application resilience challenges and evaluate the potential for GVR to broaden and simplify application resilience.

Approach

- Cross-layer Error Handling

```
main() {
    simulation_loop() {
        computation();
        if detects L1 cache parity error
            set flag = true;
        /* At designated rally point,
            each task check the flag */
        if rally point {
            if flag == true {
                roll back;
                continue;
            }
        }
        /* snapshot state periodically */
        if checkpoint point
            snapshot state
    }
}
```

![Figure 1. ddcMD x-layer Error Handling (original)](image1)

![Figure 2. ddcMD x-layer Error Handling Pseudo Code (original)](image2)

The original ddcMD is designed with resilience scheme to tolerate the primary failure mode, L1 cache parity error on BG/L. ddcMD employs a checkpoint/rollback scheme and utilizes application-level error recovery strategy, shown in Fig. 1. It periodically takes a fast checkpointing of the full computation state in memory. When an unrecoverable parity error is detected, the error handler sets a global flag. The application continues execution until it reaches a designated rally point, at which all tasks check the error flag, discard the current results, and restore to the previous backup state. The pseudo code is shown in Fig. 2. We replicate these recovery capabilities by only adding 310 lines of GVR library calls to original 11,000 lines of code, shown in Fig 3. GVR protects the essential data structures in ddcMD by creating versions of data structures periodically. Errors captured by the error detectors are exposed to application. The application then decides how to handle errors.

- Application-semantic Error Detection and Recovery

GVR enables flexible application-level error detection, which broadens the class of errors that can be detected and recovered, wherever these errors come from (L1 cache, memory, bus, network, software, etc.). One example of application-semantic error detection for ddcMD is “total energy change threshold”. Since the simulation system of atoms becomes stable after running a period, the total energy shows only tiny changes (1e-6) between two simulation time steps. Therefore, a remarkable change in total energy indicates errors. GVR also enables flexible recovery strategies. Applications can decide when and how to recover. With multi-version mechanism of GVR, applications may even recover latent errors from previous good version.
Results

Figure 4 shows that GVR-augmented ddcMD successfully recover under errors.

- Only added 310 lines to original 110,000 lines of ddcMD source code.
- GVR-augmented ddcMD can tolerate a broader class of errors than just L1 cache parity errors, such as memory errors, network errors, and software bugs, as long as these errors can be detected.

Impact

- GVR can be easily applied to large-scale applications to make the application resilient.
- GVR helps to expand existing error-tolerance apparatuses to tolerate a broader class of errors.
- Programmers can flexibly control error detection and recovery strategies that exploit application semantics.

Acknowledgements

We acknowledge our collaborators at LLNL: Ignacio Laguna and David Richards

Reference

- Aiman Fang and Andrew A. Chien, Applying GVR to Molecular Dynamics: Enabling Resilience for Scientific Computations, TR-2014-04, CS Department, Univ. of Chicago
Reliable Finite Element with GVR

http://gvr.cs.uchicago.edu/

Background
MiniFE is one of the suite of Montevo miniapps (Heroux, et al., 2009). MiniFE is a proxy for a class of applications that require an implicit solution to a set of nonlinear equations. In particular, miniFE simulates solving an unstructured grid problem with finite element method. A large proportion of computational time is spent inside a linear solver kernel— in particular, Preconditioned Conjugate Gradient method (PCG).

Finite element solvers have two primary phases of computation. The first phase generates a system of linear equations to solve based on the decomposition of the domain and the problem to be solved. The second phase solves the system of linear equations. Making the first phase fault-tolerant is a different problem than making the second phase fault-tolerant. We focused on making the second phase fault-tolerant (Rubenstein, Fujita, Zheng, & Chien, 2013) and left the first for future work.

GVR-Powered Resilience

- **Multiversion**: A GDS object can be versioned (user-defined persistent snapshot), and these persistent copies are used in error recovery. We preserve critical PCG variables using multiversioning. If an error persists across a checkpoint, we can restore from an older version.
- **Multistream**: Each GVR-protected array can be versioned at its own rate. For example, we could take snapshots of the vectors in PCG’s 3-term recursion every iteration, while taking only one snapshot of the read-only matrix A in initialization.
- **Open Resilience**: The GVR library provides a unified interface for checking for, signaling, and recovering from errors originating in either the application, or in different parts of the underlying system (runtime, OS, or underlying hardware). For miniFE, we could use GVR to expose application-detected errors or inform the application of hardware errors.
- **Trilinos Synergy**: The Trilinos project is a C++ library that provides scalable primitives for linear algebra operations, linear and nonlinear solvers, and other useful scientific computing algorithms (Heroux, et al., 2005). Trilinos hides details of communication and implementation of primitive algorithms while providing the user classes like sparse matrices and vectors. We take advantage of the abstraction that Trilinos offers by building GVR-provided resilience into linear algebra primitives rather than requiring the application developer to interact with GVR directly. We decorated Trilinos vector objects with methods to snapshot and restore state on demand with GVR (Rubenstein, Fujita, Zheng, & Chien, 2013). These methods were then used in conjunction with application-directed error detection in order to find errors and restore to a previous application state as appropriate.

Objectives

- We make the linear solver phase of miniFE (PCG) tolerant to memory corruption by taking multiple versions of key data structures and detecting errors accurately.
- Experimented with different detection and correction schemes:
  - Residual-based detection method (negligible extra work)
  - Algorithm-based detection method (extra 50% linear algebra)
Variations of snapshot/restore to correct

**Impact**
- Demonstrated that PCG can be made error-tolerant at a tolerable overhead with minimal code-change using GVR.
- Found the detector that gives the best trade-off for accuracy vs. runtime overhead.

**Results**
- Demonstrated that PCG can be made error-tolerant at a tolerable overhead.
- Residual-based error detection is not viable
  - To achieve recall of 90%, false positive rates increase to one false positive per iteration.
- Algorithm-based error detectors for PCG are more expensive but are viable
  - Achieves a recall of over 95% while expecting a false positive on fewer 1/5 of the iterations.
- Algorithm-based detection costs can be small, averaging 25% of the basic PCG computation cost (excluding any versioning overhead).
- GVR can be added to a applications conveniently:
  - Modified 3K/390K lines of code in TrilinosTpetra+underlying framework (< 1%).
  - Modified 23/3120 Lines in PCG application.

**Future Efforts:**
- Specialized detection and recovery per data object, corruption of other data objects, and scaling up.

**Acknowledgements**
We acknowledge our collaborators at Sandia: Mike Heroux, Mark Hoemmen, and Keita Teranishi.

**References**
GVR-Enabled Resilient OpenMC

Nan Dun, Hajime Fujita, Andrew A. Chien University of Chicago; John R. Tramm, Andrew R. Siegel Argonne National Laboratory. 
http://gvr.cs.uchicago.edu/

Background
Monte Carlo (MC) methods have shown a number of potential advantages over conventional deterministic methods in carrying out nuclear reactor core simulations [1]: the capability of simulating arbitrary geometrical and physics complexity, no approximation for neutron energy dependence, and inherent extreme parallelism for modern HPC architectures. However, there are still several major challenges that prevent MC methods from being a realistic choice for full-core simulation: 1) the enormous computational effort required to achieve acceptable statistics and source convergence, 2) excessive demand of memory due to large cross section (>300GB) and tally data (>1TB), 3) frequent and latent errors on exascale machines. In particular, GVR can help 2) and 3) by using versioned distributed arrays to decompose and store tally data. Current tally accumulation approaches include either simple data replication, or are based on application-controlled decomposition such as domain partitioning or client/server models, which are limited by either memory cost or performance loss [2].

How We Apply GVR
By using global addressable distributed arrays, tallies are naturally partitioned into small globally addressable blocks that fit in the limited on-node memory of compute nodes. It also greatly simplifies programmability compared to application-controlled approaches.

Multi-version is especially efficient for protecting accumulate-only tally data in MC methods from latent errors by leveraging the MC application semantics [4, 5]. Unlike checkpointing/restart or Containment Domain in which errors are corrected by rollback (i.e., correct computation is wasted), multi-versioning preserves previous correct computation, removing only the bad contribution and compensating by a partial re-computation. This approach minimizes the recovery overhead.

Impact
• A new recovery scheme for Monte Carlo methods by using forward error correction and partial re-computation.
• A new record of scaling tally accumulation with constant memory overhead.
Results

- GVR can be added to applications conveniently: Modified 300/30K lines of code in OpenMC [3] (<1%)
- Results show that the RMA-based global view array implementation is able to achieve good scaling up to 256 processes. The non-versioning distributed array achieves over 95x speedup at 256 processes, which is superior to existing published results [6].

Acknowledgements

We gratefully acknowledge the computing resources provided on Midway, high-performance computing cluster operated by the Research Computing Center at The University of Chicago.

References

GVR-Powered Resilient Trilinos

Zachary Rubenstein, Ziming Zheng, Hajime Fujita, Andrew A. Chien University of Chicago;
James Dinan Intel Corp.
Keita Teranishi Sandia National Laboratory
http://gvr.cs.uchicago.edu/

Objectives
In order to compute at exascale fault rates, application developers will have to modify existing applications. With GVR, we aim to minimize the amount of code change required in order to make an application fault-tolerant. One particularly appealing approach is to use GVR to add resilience at the library level rather than the application level so that the application can take advantage of resilience in the library rather than having to implement resilience on its own. In this study, we infuse the Trilinos scientific computation library (Heroux, et al., 2005) with GVR so that existing applications built on Trilinos can benefit from GVR-provided resilience without requiring the addition of a large amount of additional code. We also demonstrated GVR-powered Trilinos by building two resilient applications—a preconditioned conjugate gradient solver and a general minimal residual solver—on top of Trilinos.

Background
The Trilinos project is a C++ library that provides scalable primitives for linear algebra operations, linear and nonlinear solvers, and other useful scientific computing algorithms. Trilinos hides details of communication and implementation of primitive algorithms while providing the user classes like sparse matrices and vectors. Trilinos also provides common abstract solvers and preconditioners.

Preconditioned conjugate gradient is a common way to iteratively solve the linear system Ax=b. In addition, it is the simplest of the class of Krylov subspace solvers which solve linear systems by moving the approximate answer in one dimension of Krylov subspace at a time.

Like PCG, Generalized Minimal Residual Method (GMRES) is a Krylov subspace method for solving systems of linear equations. A variation of GMRES that is particularly interesting to the realm of fault tolerance is Flexible GMRES (Saad, 1993) (FGMRES) or the similar Fault-Tolerant GMRES (FTGMRES) (Hoemmen & Heroux, 2011). In this variation, about 90% of execution time is spent in the inner solver (Zheng, Chien, & Teranishi, 2014) that does not need to return the correct result. Consequently, we can afford to employ light-weight fault-tolerance methods on the inner solver and employ more heavy-weight fault-tolerance methods on the outer solver, and still converge to correct results with good performance.

How We Apply GVR
For PCG, we can take advantage of the abstraction that Trilinos offers by building GVR-provided resilience into linear algebra primitives rather than requiring the application developer to interact
with GVR directly. We decorated Trilinos vector objects with methods to snapshot and restore state on demand with GVR (Rubenstein, Fujita, Zheng, & Chien, 2013). These methods were then used in conjunction with application-directed error detection in order to find errors and restore to a previous application state as appropriate.

For GMRES, we used GVR to preserve critical data structures in FGMRES and restore them in the event in the event that an error was detected (Zheng, Chien, & Teranishi, 2014). One scheme used GVR to preserve multiple versions of critical objects during the inner solve so that, if an error was detected after the completion of an inner solve, the inner solver could be resumed from the version before the error occurred rather than having to restart the entire inner solve.

Impact

- Enables middleware developers to provide resilience while placing a minimal burden on application developers.

Results

- GVR can be added to a applications conveniently:
  - Modified 3K/390K lines of code in Trilinos Tpetra+underlying framework (< 1%).
  - Modified 23/3120 Lines in PCG application.

Future Efforts

- Taking more comprehensive advantage of GVR, such as X-layer error handling

Acknowledgements

We acknowledge our collaborators at Sandia: Mike Heroux, and Mark Hoemmen.

References


