Final Report, "Exploiting Global View for Resilience"*

Andrew A. Chien, University of Chicago and Argonne National Laboratory
Pavan Balaji, Argonne National Laboratory

Final Report DOE-UCHICAGO-SC0008603
Award DE-SC0008603 and Contract DE-AC02-06CH11357
Reporting Period: Sept 1, 2012 - August 31, 2016
March 29, 2017

1 Executive Summary

The GVR project aims to create a new approach to portable, resilient applications. The GVR approach builds on a global view data model,, adding versioning (multi-version), user control of timing and rate (multi-stream), and flexible cross layer error signalling and recovery. With a versioned array as a portable abstraction, GVR enables application programmers to exploit deep scientific and application code insights to manage resilience (and its overhead) in a flexible, portable fashion.

We have established the GVR model as a viable gentle-slope path to exascale resilisnce [3, 4]. In five demonstrations on application code bases ranging from 30K to 500K lines of code, drawn from co-design center applications and full-blown DOE science applications, we have shown that GVR can be used to add flexible resilience with modest code change (generally $< 1\%$ code change) and modest performance impact ($< 1\%$). Furthermore, this small added structure creates a sound foundation to scale from today's low error rates to the high potential error rates, and growing breadth of potential hardware and software of the future. In short, a robust path for resilience to Exascale systems and applications.

Our study of versioniong approaches, detail several low-cost methods to achieve frequent versioning. And, experiments with burst buffers, both to reduce versioning costs even further, and to engage lifetime management all show how efficient GVR versioning can be. Specific accomplishments include:

- Define GVR Model - api, and use (include versioning, multi-stream, and Open resilience) [24, 25]
- Latent Error modelling (bonus) [18]
- GVR Software - architecture, implementation, and research on key implementation challenges (efficient versioning) [2, 7, 14]

*Funded by Ofce of Advanced Scientic Computing Research, Ofce of Science, U.S. Department of Energy, under Award DE-SC0008603 and Con- tract DE-AC02-06CH11357.

- Extensive application studies with mini-apps to assess design and resilience opportunities [10, 22]
- Extensive application studies with full-applications to assess design and resilience opportunities [10]
- In-depth study with rich Trilinos libraries using GVR with full-applications to assess design and resilience opportunities [26]
- Work on ULFM which will enable GVR to be used to tolerate process failures [2]
- Extensive application studies with full-applications to GVR versioning costs [3, 4],
- Extensive implementation studies of versioning techniques, using OS support, application information [12, 13],
- Versioning optimization using burst buffers in large-scale production systems, and even lifetime management of SSDs [8, 9, 11].
- Extensive studies of application studies with full-applications to GVR versioning costs [3–6]
- Definition of a new approach for application-based fault resilience, Space-Time, that enables a disciplined approach to runtime correction/adaptation, and continuous execution. This enables creation of systems support for efficient parallel implementation.
- Partner software releases (2014), Open Source releases (2015, 2016) - see `http://gvr.cs.uchicago.edu/`
- Publication of 16 research papers and technical reports, and one two MS Thesis (Rubenstein, Fang) [2–14, 18, 22, 26]

With the one noted bonus exception, these accomplishments correspond directly to planned project objective and deliverables.

Additional project information can be found at `http://gvr.cs.uchicago.edu/`.

2 Establish GVR Model: API, Semantics, and Usage

2.1 Multi-version, Consistency, and Recovery

GVR provides a globally-visible, distributed array to applications, as in Global Arrays [19]. One of the most novel features in GVR is that it provides multi-version global array. As the computation evolves, the contents of the array keeps changing. GVR preserves multiple snapshots of these array contents, and provides random access to these snapshots. Multiple versions allow applications to enable various correction or approximation techniques to recover from complex errors.

Snapshot creations are controlled by applications – applications control when snapshots are safely taken. This also means that applications control consistency. Applications are supposed to take a snapshot at the moment where the contents of the array becomes consistent across all the users (e.g. processes) of the array. Although applications tell the library when to take a snapshot, we defined that the GVR library could ultimately decide whether to preserve that version in a memory or storage. Also the GVR library can dispose an old version if it seems unlikely to be useful anymore. In this way GVR can provide a portable runtime environment for applications, because the optimal number of versions to preserve heavily depends on several environment-dependent factors such as available memory/storage resources as well as expected failure rate.

GVR provides several APIs to create and navigate multiple versions. *GDS_version_inc* creates a version and increments the current version number. *GDS_move_to_prev* and *GDS_move_to_next* updates an array handle so that it points to previous/next version.

2.2 Multi-stream

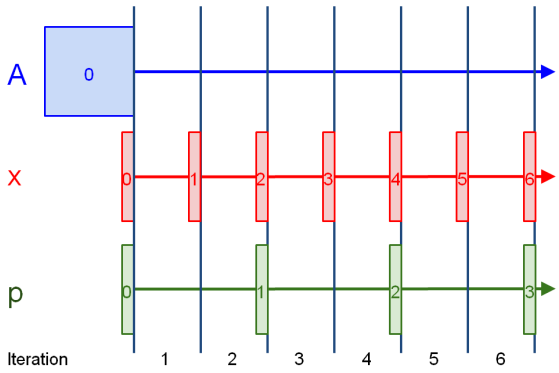


Figure 1: Multi-stream versioning in the Preconditioned Conjugate Gradient method (PCG) (each rectangle is version). The key structures are versioned at different cadences. The A matrix immutable, so only a single version is created. The x and p vectors are smaller and change each iteration, so we use periodic versioning with a higher frequency for x, customized to the PCG algorithmic error sensitivies.

One key feature of GVR is that every GDS object is independently managed. The applications can choose when to perform synchronization operations, when to take new versions, and how to handle errors on a per-object basis. This fine-grained error handling is important because different objects have different resilience characteristics. For example, some objects might be read-only, some objects might be difficult or easy to recalculate, or some objects may take up a lot of space in memory or a little space in memory. All of these characteristics imply particular ways that these objects should be made error-tolerant. For example, a read-only object needs only be preserved once, an object that is easy to calculate may not need to be preserved at all, and objects that take up less memory can be efficiently preserved at a greater frequency than objects that take up more memory.

In Figure 1 is an example of how we might preserve some of the objects in an implementation of Preconditioned Conjugate Gradient method. The large, read-only A matrix is only preserved once, while the smaller vectors are preserved more often. Of the vectors, the more vulnerable x vector is preserved more often than the less vulnerable p vector.

2.3 Open reliability – Cross layer, Unified Error Events

Program execution can encounter a variety of errors – a node crash, memory error, network error, sanity check error in a library, etc. To maximize the range of errors that an application can survive, GVR enables the use of hardware system, runtime, application, or even domain semantics. GVR provides a unified interface for error signaling and handling interface, enabling a single application

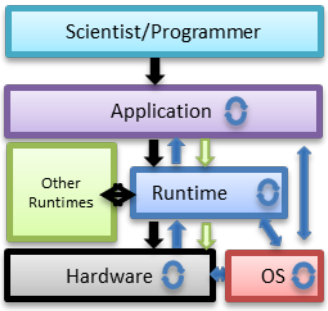


Figure 2: Open resilence supports cross-layer partnership for error recovery. Unified error types and signalling/recovery mechanisms aid portability and generality of application investment. Blue arrows between denote error signaling; black arrows error recovery. Blue circles reflect that components may handle errors internally.

handler to manage multiple classes or errors and customize that handling to application semantics. When an error happens, the GVR library invokes the appropriate handler, passing an error descriptor object with error information. The types of these objects organize an error taxonomy, making it convenient for applications to create custom (and general error handling logic. Figure 2 illustrates this cross-layer partnership for error handling.

This year we have designed and implemented the error handling APIs, and used them to implement a range of L1 cache, DRAM error, application-signaled errors, and other error handling scenarios. Future work will include additional system software error signal support.

2.4 Latent Error Modeling work (BONUS work)

While not proposed as part of the original project, we explored the implications of latent errors – commonly called "silent errors" – on the effectiveness of checkpoint-restart based reliability, comparing it to version-based resilience under a variety of assumptions of error rate, error detection rate, as well a version/version cost and recovery cost. These studies were based on a queueing theory analytical model and a set of simulation experiments.

Our work begins with the notion that errors may be latent for signicant period- s of time (in the limit truly silent), and develop a new system model for selecting checkpoint intervals in this new environment. Our multi-version scheme comple- ments this model, persisting multiple versions to enable recovery from latent errors. We use the new system model to explore the design space for future multi-version systems, characterizing opportunities to increase resilience in this new world, and how to do so how many versions might be fruitful, critical error detection and error rates, version cost, etc. These results not only shows that multi-version checkpoints increase error resilience, but that they may do so in realistic cost and error scenarios.

Our specific contributions and findings in this work include:

1. Definition of a system model that models latent errors and derivation of optimal checkpoint intervals.
2. Study of the new system model which shows that in a range of potential error rate and detection scenar- ios, 2 versions of checkpoint will be needed to achieve acceptable error cov-

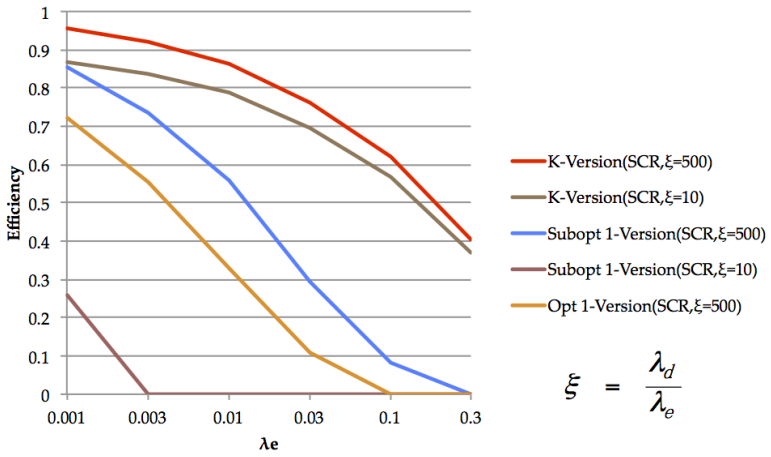


Figure 3: In presence of latent errors, K-version delivers much higher system efficiency than either SCR and retuned Daly Checkpoint restart.

erage, the specic coverage ben- ets of each additional checkpoint, and over a dozen checkpoint versions in more extreme cases.

3. Study of the eciency of multi-version shows 3 versions are generally sucient to achieve high eciency. But ten versions can be necessary to achieve high eciency in low detection rate scenarios, even with low error rates.
4. A comparison of our results to those based on a sim- pler fail-stop error model, showing that such models cannot achieve acceptable eciency in low detection rate or high error rate scenarios. In fact, the sim- pler models appear to overestimate the capabilities of single-checkpoint approaches.
5. Explore realistic exascale machine scenarios for error rate, and show that even with improved checkpoint- ing, multiple versions may give signicant benets. As many as seven checkpoints are benecial even in low error rate scenarios.

The full results [18] were published in FTXS '13.

3 Software Architecture and Implementation

3.1 Efficient Versioning

A critical challenge for the GVR is achieving efficent implementation of multi-version arrays. We are exploring several approaches. The first implements arrays as "logs" of changes, and the second uses hardware and operating system change tracking techniques.

One approach we have been developing builds a log-structured representation, capturing a sequence of modifications as they're made, so that when a new version is needed, its essentially already created. Each update is appended to an in-memory log, allowing more efficient representation if only a fraction of the array is modified in each version, increasing efficiency and reducing the size of the version. Our studies show that such is true for several application/data structures such

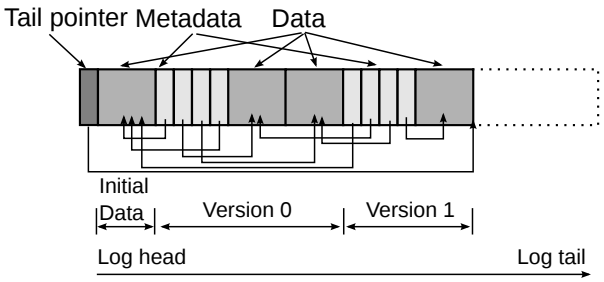
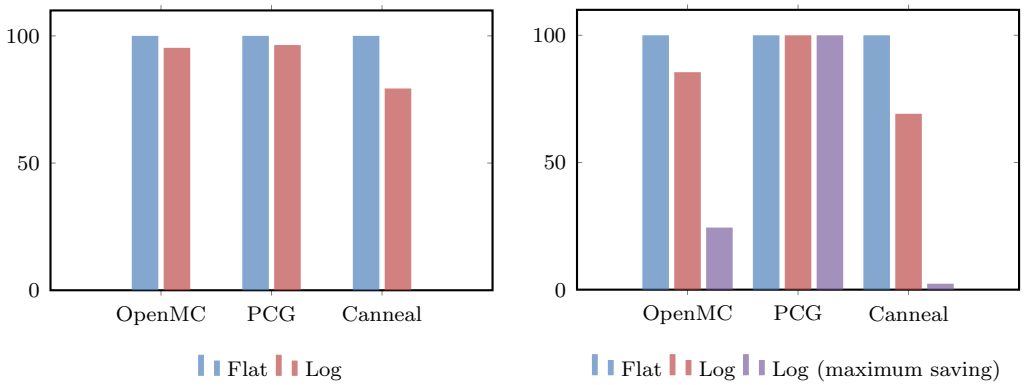


Figure 4: In-memory Data Structure of the Log-structured Array



(a) Log-array performance vs. Flat array.

(b) Version size (memory) vs. Flat array. "Log" is configured for the best runtime performance while "Log (maximum saving)" is configured for maximum memory savings.

Figure 5: Evaluation results of the log-structured array

as canneal (a part of the PARSEC benchmark suite [1]) or OpenMC tally data structure. GVR creates versions with a *GDS_version_inc()* call.

Global arrays are implemented by dividing them chunks that are assigned to nodes/processes. Each chunk has internal structure as shown in Figure 4 and is exposed to other processes via one-sided remote memory access (RMA) operations. The log area consists of tail pointer, data blocks, and metadata blocks. The tail pointer points to the tail of the log. A data block contains application program data. Metadata blocks work as an index, pointing to the corresponding location of the data block. When a new version is created, the GVR library creates a new set of metadata blocks, copying them, and writes them at the tail.

Because we are only interested in high performance implementations, we assume RDMA (Remote Direct Memory Access) hardware, and carefully design the log-structured array so that most of the fundamental data access (e.g. put/get/acc) can be performed only using one-sided communication. For example, we chose fixed-size block in order to make it easy to address the metadata block from the remote side. To optimize the performance, if the target data block is already updated, succeeding updates to the same block are overwritten to the existing block.

We applied the log-structured array to several applications (OpenMC, a Monte Carlo simulation code [20] for nuclear reactor simulation, PCG, a linear equation solver with the Preconditioned

Conjugate Gradient method using Trilinos [15], and canneal, a simulated annealing code in the PARSEC benchmark suite [1]), comparing to a flat array representation.

Figure 5 shows the runtime performance and memory usage of each application on 32 nodes. Figure 5(a) shows that versioning runtime overheads can be negligible (3.7% for PCG, 4.7% for OpenMC), and manageable for the other (26% canneal). The block size of the log-structured array was configured so that it can achieve best runtime performance. Our results show that the log-structured array achieves a comparable performance for all three applications.

Figure 5(b) shows the relative memory consumption of each application. We also measured the maximum memory savings available when the block size of the log-structured array is set to the message size. The result shows that the log-structured array saves as much as 97.7% memory for preserving versions. This means that the log-structured array can extend the lifetime of NVRAM devices which have write endurance limitations, by up to 42.7x, if versions are written to NVRAM. Full results for the log-structured array study can be found in [14].

3.2 Operating System and Hardware-based Change Tracking for Versions

Low-overhead identification of memory areas modified by the application is critical to efficient versioning. Such "change tracking" or "dirty bit tracking" is critical for applications where a non-negligible part of memory is not dirtied between version snapshots. A number of approaches can be employed to achieve that goal. The most basic scheme involves making a complete memory copy of the multiversioned memory region and comparing against it byte-by-byte at the next version increment. For applications with large memory footprint, however, the runtime overhead would be significant and the memory overhead would be unacceptable.

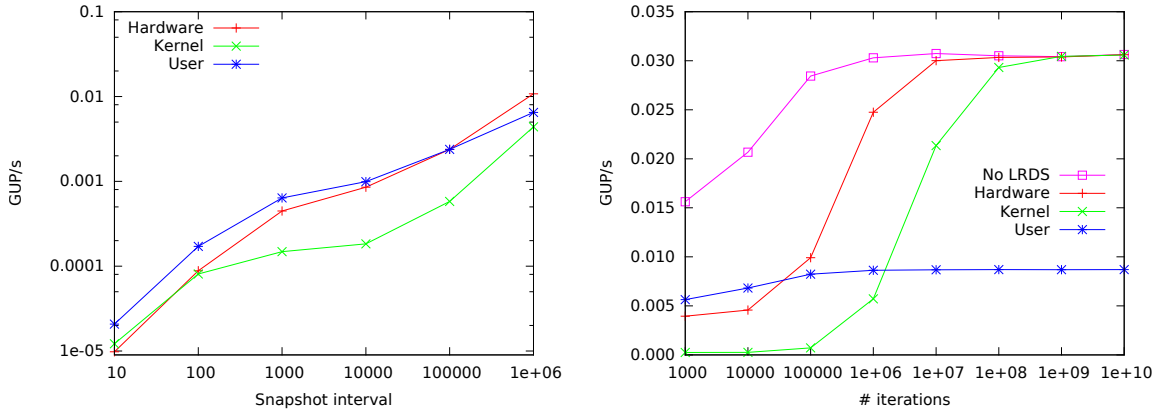
In GVR, we designed three approaches for managing change tracking between versions: (1) user-supplied dirty-bit tracking, (2) kernel-level page-based memory tracking, and (3) hardware accelerated dirty bit tracking.

User-supplied Dirty-bit Tracking. The most basic *practical* approach for dirty-bit tracking involves maintaining a bitmap for tracking changed memory areas based on user-provided hints. Being entirely userspace based, this has a relatively low overhead; however, the overhead accumulates with increasing memory access count as, in the worst-case scenario, *every* access to the multiversioned region must be registered. The bitmap resolution (block size) is a compile-time option that currently defaults to 64 bytes per bit, or a single 64-bit word per standard 4K memory page.

Kernel-level Page-based Memory Tracking. The second approach we investigated is based on kernel-level, page-based memory protection. On version increment, the multiversioned region is write-protected, subsequently resulting in page faults on write accesses. Our signal handler marks the faulting page in the bitmap as changed and unprotects it. The advantage of this scheme is that it is transparent to regular memory accesses from the application – the application code does not need to be instrumented. There are, however, a few disadvantages to this approach. First, if a write-protected region is passed to a memory-modifying system call, such as `read(2)`, the call will fail. Thus, the memory must be unprotected before the call. Second, the overhead of a page fault can be quite high. Since we use the page fault only to mark dirty pages, only the first call after a version increment has this overhead; subsequent writes are overhead-free. But depending on how frequently the versions are incremented, this overhead might or might not be amortized. Third, the resolution of change tracking is at the granularity of a page size. This is typically 4 KB, forcing

change tracking at a smaller granularity not feasible with this approach. Furthermore, for systems with large pages, the granularity might be too coarse to be valuable.

Hardware Accelerated Dirty-bit Tracking. The third approach we investigated is based on hardware dirty page bit tracking. Unfortunately, this feature is currently only available on the ia64 and x86 architectures, but we fully expect more future architectures to support similar functionality as well. This approach has most of the advantages of the kernel-based memory protection discussed above, without the overheads and limitations of page faults. Specifically, the updates of the dirty bits stored in the page tables are carried out independently by the CPU itself without any software intervention. Such capability makes this approach completely transparent to software (with the exception of (R)DMA transfers). A minor issue with this approach is that the dirty bit information is not conveniently exposed by operating systems to userspace, and no interfaces are provided to reset the page table dirty bit (which we need to do on version increment). Hence, modifications to the operating system are needed. We took advantage of the work of NCSU's team led by Frank Mueller, which in turn was based on an earlier work by HP for Itanium. We updated the patches to the latest Linux kernel version and added missing support for regular and transparent huge pages.



(a) Performance for different versioning intervals (expressed as a number of application iterations between snapshots).

(b) Performance with versioning disabled, for different total number of application iterations.

Figure 6: Performance comparison of different change-tracking schemes.

We conducted a series of experiments to compare the three approaches under different conditions and even on different hardware; Figure 6 provides a sample. The benchmark used here is the RandomAccess kernel from the HPC Challenge suite – a random walk over a 128 MB memory buffer (32K memory pages), versioned to a 1 GB ring buffer in DRAM (to avoid the highly variable overheads of I/O to more permanent storage). The experiments were run on dual quad-core Xeon Nehalem E5520 node.

The left plot shows the overall performance of the benchmark (in Giga-Updates Per Second) when varying the change tracking scheme and the frequency of versioning (from once every 10 random walk iterations to once every million). As can be observed, the first approach ("User") performs surprisingly well over a wide range of versioning intervals. In comparison, the second approach ("Kernel") performs rather poorly, being the worst or close to the worst performer over the whole range shown. This is, of course, due to the high cost of a page fault on first access to each

page (measured in a separate set of experiments at around 7000 CPU cycles). While the percentage overhead reduces when consequent versions are more than 100,000 memory accesses apart, the performance of this approach even with a million interations between versions is underwhelming. The third approach ("Hardware") reduces the overhead to some 450 CPU cycles per first page access. This significantly reduces the overhead, but still needs around 100,000 iterations between versions to hide the extra cost.

These findings are confirmed by the plot on the right, which compares the performance of the three approaches with change tracking enabled, but memory versioning disabled. Here, the X axis shows the overall number of iterations. As can be observed, the benchmark must run for over 100,000 iterations for "Hardware" to overtake "User", and for over 1 million for "Kernel" to do the same. The plot also includes a line showing the peak performance of the benchmark with change tracking disabled ("No LRDS"). Both "Hardware" and "Kernel" eventually reach that performance level, but the "User" approach never does since its overheads increase linearly with the number of iterations.

A second source of overhead for "Kernel" and "Hardware" is the system call overhead experienced when resetting the change tracking after each versioning operation. That cost is in fact a bit higher for our "Hardware" implementation than for the "Kernel" one, hence the lower performance of Hardware for the highest versioning frequency of every 10 iterations.

Hardware dirty-bit tracking on page tables is a somewhat obscure feature that is not extensively used by the mainstream operating systems, so its overheads are rather poorly documented. From what we were able to ascertain, it is implemented in the CPUs using locked read-modify-write cycles, which are fairly expensive. Similarly, the explicit reset of dirty bits that we need to conduct on version increment requires a TLB flush, resulting in another temporary slowdown. We have observed significant differences in the overhead between Intel and AMD CPUs, and even between different generations of CPUs from the same manufacturer.

Our overall conclusion though is that the "User" approach cannot be beaten in case of frequent versioning, although it does require the application code to be instrumented. The "Kernel" approach suffers from high overheads and is not as transparent as we initially thought, so it should best be avoided. Instead, the "Hardware" approach should be used in situations where instrumenting the application is impractical and the multiversioned memory region is large, provided that it is possible to run a custom operating system kernel on the machine. The memory access pattern of the application must also be sufficiently localized that the same memory pages are written to on average at least some 3 times between versioning operations in order to hide the extra overhead of the first access.

3.3 Tolerating Process Failures

As fault tolerance has become an increasingly important area of research in the push toward exascale, it has become just as important to be able to evaluate the effectiveness and performance of new fault tolerance techniques with tools that are efficient, representative of real-world applications, and easy to modify to demonstrate computer science research. The CESAR mini-apps provide such a tool which we have used to evaluate two new libraries which provide fault tolerance in MPI applications. By using these apps, we have been able to quickly refine the correctness and performance of fault tolerance libraries in relation to the GVR project.

The first such library is MPIXFT, an MPI-3 based library which provides automatic fault tolerance, transparently detects and recovers from process failures. MPIXFT captures MPI calls

via the PMPI interface and translates them to use an underlying communication mapping that can be dynamically regenerated when a failure is detected. By converting the application's MPI calls into new calls, the library can use its own MPI objects, such as communicators, remap ranks to match the new communicators, and convert blocking calls into non-blocking calls to employ failure propagation.

Using Mira and other, smaller clusters, we are able to determine the efficiency of our methods at a more representative scale of the type of machine where such failures are expected to occur. To effectively reach such a scale, we modified the Monte Carlo Communication Kernel (MCCK) mini-app to recover from process failures by re-executing failed iterations before committing the results to memory. MCCK is a domain decomposition style application that uses 3D halo exchange to move particles between domains. In our modifications, we check whether an MPI call was successful by examining its return code. If a call is unsuccessful, then the result of that iteration is discarded and the entire iteration is repeated (after running to completion to avoid improper MPI call matching). This means that before the transferred particles are committed to memory and the previous particle counts are overwritten, the counts are stored in a second buffer which then replaces the original buffer after all of the MPI calls return successfully. Another important modification to the MCCK mini-app is to allow the application to complete with fewer processes than what it began with. When a process failure occurs, MPIXFT does not replace the failed process, but allows the application to easily avoid the failed process in future communication. We modified the MCCK mini-app to divide the failed process's domain amongst its 6 neighboring processes. Particles which would have been sent to the failed process are instead sent "through" the process to the next neighbor in the communication topology.

Another fault tolerance library that has benefited from the CESAR mini-apps is the MPI library itself. The MPI Forum is developing the next version of the MPI Standard, and as part of that effort, is evaluating how fault tolerance in MPI might be specified. The current proposal under consideration is called User Level Failure Mitigation (ULFM) [2]. This proposal puts failure recovery in the hands of the application (or libraries which work on its behalf) by providing the foundational API calls necessary to stabilize the MPI library and reconstruct any communication interfaces. ULFM specifies the general class of failure detector that is required and the interaction between the library and application in order to report such failures to the application in a timely and efficient manner via return codes and error handlers. After reporting such a failure, the application can use failure discovery APIs to query the group of failed processes and communicator object reconstruction calls to re-create MPI communicators, windows, and files. ULFM also includes a new type of API called MPI_COMM_REVOKE which allows the application to enforce global knowledge of failures by destroying an exiting communication object. The API for ULFM has been defined by the MPI Forum's Fault Tolerance Working Group and now evaluation of its effectiveness and performance are ongoing.

While this new API targets large applications that run at exascale, more immediate evaluation requires more manageable representative applications such as those available from CESAR. Evaluation in this area is ongoing and therefore details of such work are not yet available, but the current plan is to modify one or more of the CESAR mini-apps (including MCCK and possibly Nekbone) to use such an API. These modifications will be less extensive than were necessary for MPIXFT as they will not require the application to run with fewer processes after a failure. Instead, the failed process can be re-spawned and patched back into the communication topology using existing MPI dynamic processing APIs and the new ULFM proposal.

As these libraries are completed, we have been designing interaction between them and GVR, primarily with ULFM. By combining the process resilience of ULFM and the data resilience of GVR, we can provide a more complete model of fault tolerance for applications. We have been beginning work collaborating with Lawrence Berkeley National Laboratory to add GVR+ULFM to their Chombo project (more information in Section **??**).

3.4 Storage Hierarchy and Redundant Encodings

We have explored two additional implementation opportunities. First, we have studied the integration of GVR versioning with the scalable checkpoint-restart (SCR) system. Second, we developed an in-memory redundant encoding scheme (XOR and erasure codes) that allows versions to be persisted reliably in memory. Note that this type of redundancy can be supported at reasonable cost in versions because the data does not change. All GVR versions older than the current one are immutable.

4 Application studies: Version-based Recovery in mini-Apps, co-Design Apps, and full Applications

To ensure a robust, usable API design, and to explore the utility of GVR with HPC applications, we have done studies using GVR's multi-versioning approach to add resilience to a number of applications. Some of these studies were based on released mini-apps, but significantly several involved large-scale applications and were done in partnership with a computational science team. Learnings include insights into the ease with which GVR was added, its ability to flexibly add a spectrum of resilience approaches and to adjust overhead as needed to acceptable levels. In all cases, our experience has been positive, with only modest code changes required, the resilience scheme flexibly adapted to both match application structure and exploit application semantics, and acceptable low overheads. In the following we discuss our studies of miniMD, ddcMD, miniFE, PCG with Trilinos, and GMRES with Trilinos.

4.1 miniMD

MiniMD is one of the Sandia National Laboratories Mantevo mini-apps [15], a miniature version of molecular dynamics (MD) application LAMMPS.

Like LAMMPS, MiniMD uses spatial decomposition MD, where individual processors in a cluster own subsets of simulation box and perform atoms computation with problem size, atom density, temperature, timestep size, particle interaction cutoff distance and other parameters specified by users. It also has force models for binding forces, electrostatic forces, and so on typical to MD applications. MiniMD computes the evolution of a set of particles by a series of timesteps. Each timestep includes

1. update velocity using forces
2. update position using velocity
3. build neighbor lists
4. compute force using position

5. apply constraints & boundary conditions on force
6. update velocity using new force
7. output

The state of the simulation is capted in three parameters for each of the molecules – position, velocity, and force. To make MiniMD resilient using GVR, each of these parameters is stored and versioned in a global array; three global arrays in all. Upon error detection, MiniMD simply reads the most recently saved version of each of these three arrays (position, velocity, and force), and restarts the computation. Obviously if only one array is corrupted and the application allows, a single array can be restored. Alternatively, as we see in ddcMDC forward error correction may be possible.

To exercise the resilient miniMD and GVR, and application-level error checking, we performed the error-injection experiments. Errors were injected into random locations in the position, velocity and force vectors in the runtime. Application level checks based on molecule loss (checksum), particle movement continuity, and some energy conservation constraints were explored. When errors were detected, MiniMD restored the computation and generated the correct results.

4.2 ddcMD

Domain-decomposition molecular dynamics (ddcMD) is a collection of atomistic simulation programs developed by Lawrence Livermore National Laboratory. It is designed to achieve scalability and efficiency. The existing implementation of ddcMD tolerates L1 cache parity errors on BG/L by using a "simply rally" checkpoint/restart scheme. We used GVR to broaden the fault tolerance capabilities of ddcMD.

The way that ddcMD models physics is much more complicated than the model used in MiniMD. We first analyzed the data structures in ddcMD and identified a set of variables that are essential for computation and recovery, then applied GVR global arrays to preserve these variables. We designed two error detection methods. We found that the capabilities of fault tolerance depends on error detection methods [10] since the recovery procedure could only be invoked upon errors being detected.

Furthermore, we conducted error-injection experiments in two steps. First, we injected errors into different variables. Second, we varied the magnitude of of errors. Using these different scenarios, we tested the correctness of recovery and compared the sensitivities of our error-detection methods. Application level checks based on molecule loss (checksum), particle movement continuity, and some energy conservation constraints were explored. Full details of this studay can be found in [10]:

Results Following the ddcMD code changes made to tolerate hardware unrecoverable L1 cache parity errors, we replicated these recovery capabilities with only adding 310 lines of GVR library calls to original 10,935 lines of source code. Our next step was to use this base to explore a range of application-specific error detection and recovery schemes that generalize the classes of errors that can be detected and recovered without application interruption. This broader class of errors includes general memory system errors (L2, L3, DRAM, bus, controller, etc), hardware computation errors, communication errors, software bugs, and others. The error checks are conveniently expressed in the application source code in terms of application data structures, and enable flexible, application-controlled recovery from these errors. We find that GVR enables convenient broadening of error

coverage and resilience. To evaluate the capabilities of error detection schemes, we performed error injection experiments. The results show that application-specific error detection schemes can detect certain magnitudes of errors, but leave some errors silent. Our GVR provides opportunities to recover from silent errors

4.3 miniFE

MiniFE is a Mantevo miniapp [16] that simulates solving a structured mesh 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 [22] and left the first for future work.

Results For miniFE, we used GVR to preserve critical elements of the state of computation and then restore them in the event of drastic increase in the distance between the approximate answer and the correct answer. We expanded our work with PCG in a further study, described in Section 4.4.

We performed an exploratory error-injection experiment in which we severely corrupted every element of the critical r vector in PCG. We then used GVR to periodically take snapshots of critical variables, and then, in the event of a drastic norm residual increase, restore the solver to its previous state. After an error was injected, norm residual returned to its value at the last snapshot rather than drastically growing.

4.4 PCG/Trilinos

The Trilinos project [15] is a C++ library that provides scalable primitives for linear algebra operations, linear and nonlinear solvers, and other useful scientific computing algorithms. In this study, we utilized Trilinos' linear algebra primitives in order to implement a PCG solver and expand on the work discussed in 4.3.

Preconditioned conjugate gradient is a common iterative solution method fore linear systems $Ax = b$. 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. It is not clear how errors in PCG should be efficiently detected. Even when an error is detected, there are a number of conceivable ways to recover, including restoring old state, ignoring errors and depending on numerical resilience, or replacing corrupted data with some approximation of the correct value.

Results - GVR and Effective Error Detection We exploited core Trilinos abstractions for vectors and matrices (petra) to build GVR-provided resilience into linear algebra primitives rather than requiring the application developer to interact with memory directly. We decorated Trilinos vector objects with methods to snapshot and restore state on demand with GVR [22]. 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. This GVR-enhancement of the trilinos system required changes to only a very small fraction of the Trilinos source code [26].

GVR enabled convenient expression of application-level error checks connected to application-driven recovery. Our experiments with the PCG solver show that the choice of detection methods

make a good deal of difference when correcting errors in PCG. Inexpensive methods based on monitoring the norm residual and more expensive, algorithm-aware methods that performed extra linear algebra operations to verify PCG-specific invariants were implemented and explored empirically.

We found that: 1) Though inexpensive, residual-based detection performs poorly. To achieve acceptably low false negative rates, high (30x number of mitigated false negatives) false positives rates are required. 2) Though more expensive, algorithm-based detection performs better overall, achieving much lower false negative rates at one seventh the false positive rate. Even this relatively expensive error detection is inexpensive compared to a single solver iteration, and therefore is viable for linear solvers—particularly in high error-rate systems.

4.5 GMRES/Trilinos

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 [23] (FGMRES) or the similar Fault-Tolerant GMRES [17] (FTGMRES). In this variation, each iteration of GMRES utilizes an inner solver to solve a linear system that is simpler than the system that FGMRES is ultimately trying to solve. In principal, FGMRES will eventually return correct results regardless of the results of the inner solve. In addition, about 90% of execution time is spent in the inner solver [26]. 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.

As in the PCG project, this work utilized Trilinos library. The work employed both Trilinos' implementation of linear algebra primitives and Trilinos' GMRES implementation.

We used GVR to preserve critical data structures in FGMRES and restore them in the event in the event that an error was detected [26]. 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 solver, the inner solver could be resumed from the version before the error occurred rather than having to restart the entire inner solver.

We found that, even though FGMRES is inherently resilient to inner solver errors, performance in a faulty environment could be significantly improved by utilizing GVR for fault tolerance. In addition, utilizing even very expensive Dual-modular redundancy in the outer solver significantly improves performance over restarting.

4.6 OpenMC

OpenMC is a open source production code for conducting direct full-core reactor simulation by using Monte Carlo methods [20]. OpenMC is able to scale up to tens of thousands of processors by using several novel techniques, enabling high fidelity, large-scale reactor simulations on modern and future computer systems. It was originally developed by the Computational Reactor Physics Group at the MIT in 2011 and now is being used for reactor studies at the Center for Exascale Simulation of Advanced Reactors (CESAR) at ANL.

We have been actively working with CESAR group to leverage the application knowledges to help us design and apply global view and resilience architectures to OpenMC. We found that global view and multiversion are promising alternatives to improve the scalability, programmability, and

resilience in OpenMC, and our approaches can be extended to a variety of Monte Carlo applications. To this end, we apply GVR to two major data structures in OpenMC: cross sections and tally data.

- Global view helps OpenMC decompose large cross sections (˜100 GB) and tally data(˜10 TB) that can not be fit in on-node memory in realistic simulations by data partition and distribution using global view arrays. The global view approach also provides much better programmability for particle-based parallelization in OpenMC.
- Multiversion is especially useful for protecting accumulate-only tally data in Monte Carlo methods from errors. At the end of each repeated stage of simulation, tally data is snapshotted as a version T_i, composing a history of tally data $T_1 \dots T_n$. Since the tally scoring is Monte Carlo accumulation, if one latent error happened in stage i, then we are able to recover the T_n with error to correct $T_n^{'}$ by

$$T_n^{'} = T_n - (T_i - T_{i-1}) + Recompute(batch_i) \quad (1)$$

  This approach is more efficient than checkpointing/restart because we preserve the computation effort from batch $i+1$ to batch n, while checkpoint/restart needs to roll back and start over from batch $i+1$. It also enables fine-grained recovery from multiple concurrent accumulation streams.

We have successfully applied global view approach to tally data by integrating GVR into OpenMC [7]. Our evaluation shows that the global view approach achieves better scalability than other data decomposition approaches in terms of both memory cost and parallel performance. In particular, using RMA-based global view arrays achieve 60% of efficiency of ideal scaling and scales well up to 256 processes (see Figure 7). Note that global view approach without buffering gains at least 1.3× speedup than another data decomposition approach using tally servers. In tally server approach [21], among total p processes, a portion of s processes are dedicated to receiving tally scores from other $c = p - s$ compute processes. In our experimental environments, the optimal tally server ratio is $c/s = 3$ as shown in Figure 7. The loss in efficiency of tally server is mainly due to only c processes are available to simulate particles when all p processes are available for computation in global view case. Besides above preliminary performance achievement, a prototype implementation of using multiversion tally data also demonstrates the effectiveness of our resilience model.

Our next step includes: 1) implement the fine-grained recovery hierarchy for multiversion tally data and evaluate the recovery efficiency of the multiversion scheme comparing to other C/R approaches, and 2) design and implement a caching scheme in global view for read-only cross sections and compare it with other optimization for cross sections reference performance.

4.7 Summary of Application Code Change Required

We have shown that GVR can provide portable, flexible resilience applied at the code architecture level. However, for resilience code changes to be viable, they must be localizable (don't require changes in architecture), simple, and have leverage as error rates increase and recovery approaches proliferate to cover increasingly faulty hardware and software.

Our studies (summarized in Figure 8) show that in general, the GVR approach is practical for application to large-scale DOE applications and scaling forward to exascale systems. Because the required effort is incremental and small, GVR represents a viable "gentle slope" path for exascale resilience.

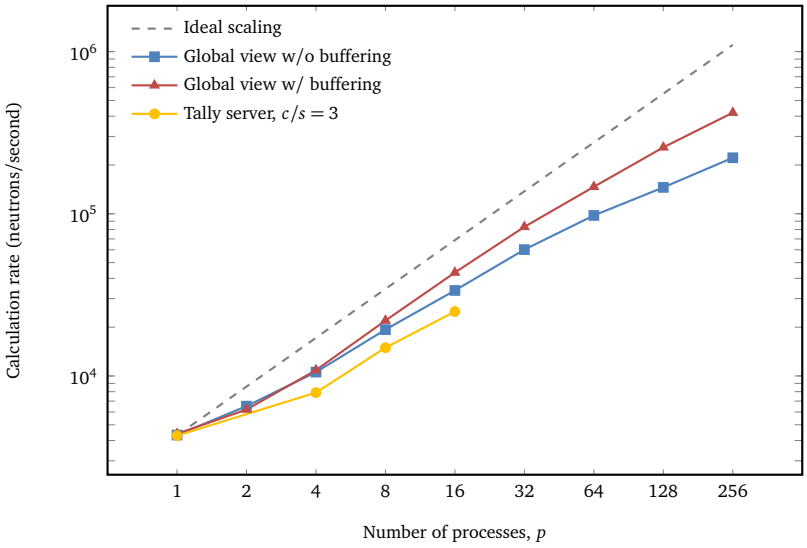


Figure 7: OpenMC performance with global view arrays for tally data decomposition. The performance scales close to linearly up to 32 processes and shows a degradation beyond. Adding a buffering technique to reduce RMA contention improves scaling efficiency by an additional 1.5x to 60% efficiency.

5 Software Releases

5.1 Partner releases

The GVR software includes a full implementation of the versioned global array API, cross-layer error signalling and recovery, and a variety of implementations of each of these API's. To date, the GVR software has been distributed in bi-lateral engagements focused on particular codes and experiments. These include:

- 4Q2013, Partner release to David Richards and Ignacio Laguna of LLNL for resilience studies of GVR with the ddcMD molecular dynamics code.
- 4Q2013 Partner release with Andrew Siegel and John Tramm of the CESAR co-design center and ANL for resilience and scaling studies of the OpenMC monte carlo neutronics code.
- 1Q 2014 Partner release with Brian van Straalen and Anshu Dubey of LBNL for resilience studies of GVR with the Chombo adaptive mesh refinement code.
- 4Q 2014 Open Source Release from http://gvr.cs.uchicago.edu/. Platforms supported include x86-64 Linux cluster, Cray XC30 and IBM Blue Gene/Q.
- 3Q 2015 Open Source Release from http://gvr.cs.uchicago.edu/. Improved scalability, support for hierarchical storage management, and compatible with SCR.

Code/ Application	Size (LOC)	Changed (LOC)	Leverage Global View	Change SW architecture
Trilinos/PCG	300K	<1%	Yes	No
Trilinos/ Flexible GMRES	300K	<1%	Yes	No
OpenMC	30K	<2%	Yes	No
ddcMD	110K	<0.3%	Yes	No
Chombo	500K	<1%	Yes	No

Figure 8: Application studies and the scale of required code changes to add GVR-based resilience. In all cases, the code changes are small and localized. In all cases, no software architecture changes are required.

5.2 Performance characterization and improvement

5.2.1 Communication

For one-sided operations over Ethernet, GVR has comparable or better performance to GA for the two-node and multiple-client distributed array access operations and for the multiple-target get operation, while GA has better performance for the multiple-target put and accumulate operations and for the high-load distributed array access operations. GVR also shows faster local access (when the data size transferred exceeds 8,192 bytes) and smaller non-blocking initiation overhead than GA.

For one-sided operations over RDMA (e.g., Infiniband), previous implementation had a non-trivial performance penalty due to the busy waiting (from implementation in MVAPICH2/IB), in communication thread. To this end, we reimplemented the message waiting loop by using a MPI_Testany() and spin approach, which significantly improved the performance. As a result, current implementation delivers comparable performance (less than 10% overhead) to native MPI one-sided operations, and thus enables good performance and scalability for applications such as OpenMC.

5.2.2 Versioning Using Burst Buffers

Resilience has become a major concern in high-performance computing (HPC) systems. Addressing the increasing risk of latent errors (or silent data corruption) is one of the biggest challenges. Multi-version checkpointing system, which keeps multi-version of the application states, has been proposed as a solution and has been implemented in Global View Resilience (GVR). The resulting more sophisticated management of data introduces overheads and the resulting impact on performance need to be investigated. In this paper we explore the performance of GVR for an HPC system with integrated non-volatile memories, namely Blue Gene Active Storage (BGAS). Our empirical study shows that the BGAS system provides a significantly more efficient basis for flexible error recovery by using GVR multi-versioning features compared to using a standard external storage system attached to the same Blue Gene/Q installation. Using BGAS especially achieves at least

10x performance boost for random traversal across multiple versions due to significantly better performance for small random I/O operations [8]

5.2.3 Burst Buffer Lifetime Management

We consider the use of non-volatile memories in the form of burst buffers for resilience in supercomputers. Their cost and limited lifetime demand effective use and appropriate provisioning. We develop an analytic model for the behavior of workloads on systems with burst buffers, and use it to explore questions of cost-effective provisioning, and mission-directed allocation of burst-buffer (SSD) lifetime.

First, our results show that system efficiency can be increased by as much as 14% by considering a global perspective (workload mix, job size) for SSD lifetime allocation. Second, with size-based and system-efficiency based lifetime allocation, large jobs suffer as much as 40% job efficiency loss; job-efficiency based allocation must increase their allocations by 50% to eliminate this disparity. Finally, further results suggest that under provisioning SSD lifetime (only 10-20% of the "optimum" as defined by per-job requirements without resource constraint) is sufficient to produce 90% system efficiency at failure rates three times that of current systems.

For more information see [9, 11]

6 Publications

References

[1] Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[2] Wesley Bland, Aurelien Bouteiller, Thomas Herault, Joshua Hursey, George Bosilca, and JackJ. Dongarra. An evaluation of User-Level Failure Mitigation support in MPI. *Computing*, 95(12):1171–1184, 2013.

[3] A. Chien, P. Balaji, N. Dun, A. Fang, H. Fujita, K. Iskra, Z. Rubenstein, Z. Zheng, J. Hammond, I. Laguna, D. Richards, A. Dubey, B. van Straalen, M Hoemmen, M. Heroux, K. Teranishi, and A. Siegel. Versioned distributed arrays for resilience in scientific applications: Global view resilience. In *Proceedings of the International Conference on Computational Science (ICCS)*, 2015. Reykjavik, Iceland.

[4] A. Chien, P. Balaji, N. Dun, A. Fang, H. Fujita, K. Iskra, Z. Rubenstein, Z. Zheng, J. Hammond, I. Laguna, D. Richards, A. Dubey, B. van Straalen, M Hoemmen, M. Heroux, K. Teranishi, and A. Siegel. Versioned distributed arrays for resilience in scientific applications: Global view resilience. *IJHPCA*, September 2016.

[5] Anshu Dubey, Hajime Fujita, Daniel T. Graves, Andrew Chien, and Devesh Tiwari. Granularity and the cost of error recovery in resilient amr scientific applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '16, pages 42:1–42:10, Piscataway, NJ, USA, 2016. IEEE Press.

[6] Anshu Dubey, Hajime Fujita, Zachary Rubenstein, Brian Van Straalen, and Andrew A. Chien. *A Case Study of Application Structure Aware Resilience Through Differentiated State Saving and Recovery*, pages 619–630. Springer International Publishing, Cham, 2015.

[7] Nan Dun, Hajime Fujita, John Tramm, Andrew A. Chien, and Andrew R. Siegel. Data decomposition in Monte Carlo particle transport simulations using global view arrays. *IJHPCA*, March 2015.

[8] Nan Dun, Dirk Pleiter, Aiman Fang, Nicolas Vandenbergen, and Andrew A. Chien. *Multi-versioning Performance Opportunities in BGAS System for Resilience*, pages 486–504. Springer International Publishing, Cham, 2016.

[9] Aiman Fang. How much ssd is useful for resilience in supercomputers. Master's thesis, University of Chicago, Department of Computer Science, 2015.

[10] Aiman Fang and Andrew A. Chien. Applying gvr to molecular dynamics: Enabling resilience for scientific computations. Technical Report TR-2014-04, Department of Computer Science, University of Chicago, April 2014.

[11] Aiman Fang and Andrew A. Chien. How much ssd is useful for resilience in supercomputers. In *Proceedings of the 5th Workshop on Fault Tolerance for HPC at eXtreme Scale*, FTXS '15, pages 47–54, New York, NY, USA, 2015. ACM.

[12] H. Fujita, K. Iskra, P. Balaji, and A. A. Chien. Empirical comparison of three versioning architectures. In *2015 IEEE International Conference on Cluster Computing*, pages 456–459, Sept 2015.

[13] H. Fujita, K. Iskra, P. Balaji, and A. A. Chien. Versioning architectures for local and global memory. In *2015 IEEE 21st International Conference on Parallel and Distributed Systems (ICPADS)*, pages 515–524, Dec 2015.

[14] Hajime Fujita, Nan Dun, Zachary A. Rubenstein, and Andrew A. Chien. Log-structured global array for efficient multi-version snapshots. In *Submitted for publication*, 2014.

[15] Michael A Heroux, Roscoe A Bartlett, Vicki E Howle, Robert J Hoekstra, Jonathan J Hu, Tamara G Kolda, Richard B Lehoucq, Kevin R Long, Roger P Pawlowski, Eric T Phipps, et al. An overview of the trilinos project. *ACM Transactions on Mathematical Software (TOMS)*, 31(3):397–423, 2005.

[16] Michael A Heroux, Douglas W Doerfler, Paul S Crozier, James M Willenbring, H Carter Edwards, Alan Williams, Mahesh Rajan, Eric R Keiter, Heidi K Thornquist, and Robert W Numrich. Improving performance via mini-applications. *Sandia National Laboratories, Tech. Rep. SAND2009-5574*, 2009.

[17] Mark Hoemmen and M Heroux. Fault-tolerant iterative methods via selective reliability. In *Proceedings of the 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC). IEEE Computer Society*, 2011.

[18] Guoming Lu, Ziming Zheng, and Andrew A. Chien. When is multi-version checkpointing needed? In *Proceedings of the 3rd Workshop on Fault-tolerance for HPC at extreme scale*, FTXS '13, pages 49–56, New York, NY, USA, 2013. ACM.

[19] Jarek Nieplocha, Bruce Palmer, Vinod Tipparaju, Manojkumar Krishnan, Harold Trease, and Edoardo Apr+. Advances, applications and performance of the Global Arrays shared memory programming toolkit. *International Journal of High Performance Computing Applications*, 20(2):203–231, Summer 2006.

[20] Paul K. Romano and Benoit Forget. The OpenMC Monte Carlo particle transport code. *Annals of Nuclear Energy*, 51:274–281, 2013.

[21] Paul K. Romano, Andrew R. Siegel, Benoit Forget, and Kord Smith. Data decomposition of Monte Carlo particle transport simulations via tally servers. *Journal of Computational Physics*, 252:20–36, 2013.

[22] Zachary Rubenstein, Hajime Fujita, Ziming Zheng, and Andrew Chien. Error checking and snapshot-based recovery in a preconditioned conjugate gradient solver. Technical Report TR-2013-11, Department of Computer Science, University of Chicago, November 2013.

[23] Youcef Saad. A flexible inner-outer preconditioned gmres algorithm. *SIAM Journal on Scientific Computing*, 14(2):461–469, 1993.

[24] GVR Team. Gvr documentation, release 0.8.1-rc0. Technical Report 2014-06, University of Chicago, Department of Computer Science, 2014.

[25] GVR Team. How applications use gvr: Use cases. Technical Report 2014-05, University of Chicago, Department of Computer Science, 2014.

[26] Ziming Zheng, Andrew A. Chien, and Keita Teranishi. Fault tolerance in an inner-outer solver: A gvr-enabled case study. In *11th International Meeting High Performance Computing for Computational Science-VECPAR 2014*, 2014.