Log-Structured Global Array for Efficient Multi-Version Snapshots

Hajime Fujita*†, Nan Dun*†, Zachary A. Rubenstein*, Andrew A. Chien*†
*University of Chicago, †Argonne National Laboratory
{hfujita, dun, zar1, achien}@cs.uchicago.edu

***Abstract*—In exascale systems, increasing error rates—particularly silent data corruption—are a major concern. The Global View Resilience (GVR) system builds a a new model of application resilience on versioned arrays. These arrays can used exploited for flexible, application-specific error checking and recovery. We explore a fundamental challenge to the GVR model – the cost of versioning. We propose a novel log-structured implementation that appends new data to an update log, simultaneously tracking modified regions and versioning incrementally. We compare performance of log-structured to traditional flat arrays using micro-benchmarks and several full applications, and show that versioning can be more than 10x faster, and reduce memory size significantly. Further, in future systems with NVRAM, a log-structured approach is more tolerant of NVRAM limitations such as write bandwidth and wear-out.**

I. Introduction

With the widely documented changes in CMOS scaling, power is an increasingly critical concern for systems from mobile to supercomputer scale. As a result, both aggressive voltage scaling and the physical challenges of deep submicron technologies (14nm and 7nm) give rise to increased error rates, wearout, and manufactured variability. Consequently, computer reliability is an increasing concern, particularly in high-performance computers whose extraordinary scale (over 1 million cores) exacerbates both power and reliability concerns [1], [2], [3], [4]. Recent studies of modern supercomputers have shown failure as the norm rather than the exception, with system-wide mean time between failures of a few hours [5], [6]. Many expect this situation will deteriorate further in future HPC systems, with exascale systems projected to have mean time to interrupt (MTTI) as low as 10 to 30 minutes [7], [8], [2], [9]. Under these circumstances, without help, applications will struggle to make efficient progress.

While many failures incur detectable data loss or process and node crashes[1], recent years have seen growing concern about *latent errors*, (silent data corruption or SDC [10]) because so many HPC modeling computations are sensitive, high precision simulations of chaotic of unstable systems. Also, numerous reports document SDC errors are more frequent than previously understood. For example, the BlueGene/L system (106,496 nodes) experiences an L1 cache error (parity error) every 4-6 hours [9]. The Cray XT5 at Oak Ridge National National Laboratory experiences an uncorrectable double bit error on a daily basis [11]. Higher error rate will also raise the probability of latent error occurrence, because for example for memory errors, multi-bit error that cannot be detected by ECC or chipkill would happen more often. Latent errors are generally invisible (silent) until the corrupted data is activated [12], so the detection intervals can be heavily algorithm-dependent or application-dependent, and generally much longer. As shown in previous studies [13], [11], the latent errors can create severe application problems such as incorrect results or extreme performance degradation. Identifying and dealing with latent errors efficiently is challenging. Memory scrubbing can uncover errors, but is expensive [14], [15]. Software solutions employ redundancy and specialize fault-tolerant algorithms [16], [17], application-level error checking [15], and critical MPI message validation [11]. However general techniques are rare, and known techniques introduce significant overhead for parallel applications [11], [18], and consequently are rarely used.

Versioned arrays are one promising general approach for identifying and dealing with latent errors. Multiple versions of array contents provides data redundancy, which allows applications to rollback from a version taken before the error [19], [20], or even apply forward recovery driven by an application-specific knowledge.

The Global View Resilience (GVR) library is designed to enable programmers to create portable applications that are resilient to a broad range of errors including traditional process/node crashes and more difficult *latent* errors. Specifically, applications designers can employ:

- multi-version global arrays (a.k.a. global data structures) (enable complex and latent error recovery),
- multi-stream versioning (allow each data structure to be versioned under programmer control), and
- unified error signaling and handling, customized for each global data structure (allow simple checks and techniques to handle large classes of error, and exploitation of application semantics).

In this paper, we focus on efficient implementation of multi-version arrays, the central portable abstraction in GVR for data resilience. Varied application studies involving linear solvers, Monte Carlo methods, particle codes, and even adaptive mesh refinement have demonstrated the GVR approach promising for both portable, flexible resilience that can handle many different hardware and software errors in an application-custom fashion [2]. However, a common concern is *what is the cost of versioning?*. For high-performance computing programs in particular, such concerns are critical to the practical viability of the GVR approach.

We propose a new approach to versioning – a log-structured implementation – and evaluate that approach with micro-benchmarks and applications.. First, we describe our design and implementation of the log-structured array, with two different

[1] As well as many correctable data errors, but we focus on those not automatically corrected by the hardware here.

[2] See http://gvr.cs.uchicago.edu/

access schemes (message- and RMA-based). A particular challenge in large parallel machines is effective use of both message passing and remote-memory access (RMA). Without RMA, competitive performance is challenging. Second, we evaluate and compare the log-structured approach to the traditional flat array approach using several micro-benchmarks and synthetic benchmarks to measure communication latency, bandwidth, and version increment cost. Finally we evaluate both log-structured and flat implementations using three full applications, OpenMC, canneal, and preconditioned conjugate-gradient. This last evaluation is done for a DRAM-only system, and a system that uses DRAM and SSD/Flash to store versions.

Specific contributions include:

- design of a log-structured implementation of arrays that supports efficient versioning and RMA access
- evaluation of versioning in flat (traditional) and log-structured implementations using microbenchmarks shows 10x faster versioning for 1MB array
- overall, the micro-benchmarks indicate that log-based implementations deliver comparable performance on reads (within 26% overhead), but incur additional overheads on writes (from 7% to 99%). Overall performance will depend on workload
- the synthetic benchmark shows that the log-structured implementation can take snapshot 19x faster than flat array on 5 puts/version workload, while saving more than 99% of memory.
- application benchmarks show negligible versioning runtime overheads in most cases (3.7% for PCG, 4.7% for OpenMC), and reduces memory for preserving versions by 30% to 48%.
- with NVRAM, log-structured approach increase tolerance of low write bandwidth or limited lifetime, improving performance by 20% (OpenMC, with SSD).

II. Background

A. Global View Resilience

The Global View Resilience (GVR) project supports a new model of application resilience built on versioned arrays (multi-version). A programmer can select a global array [21] for versioning and control timing and frequency (multi-stream). Access to these arrays is provided through dedicated library calls such as *put* or *get*. Application also controls when to create a version by calling *version_inc* collectively. The timeline of application state created by versioned arrays can then be used to both check application data for errors, and to recover from said errors (application-customized checking and recovery). While the GVR system provides consistent versions of single array, any coordination across multiple arrays (*i.e.* across the multiple streams) is an application responsibility. Old versions become read-only from the application semantics. This property allows the GVR library to transform the versions easily, such as compression, encryption, hardening with error checking code, etc. Because the GVR library operates at the level of application arrays, it is both convenient to use and portable, enabling convenient portable resilience.

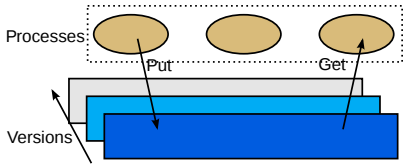


Fig. 1. Multi-version global array in GVR

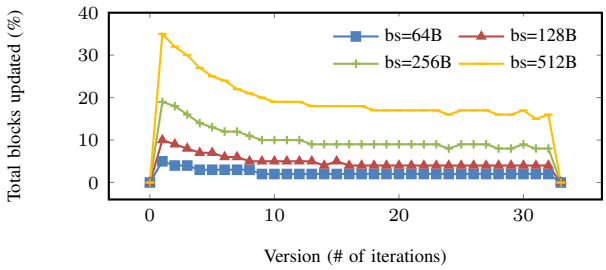


Fig. 2. The canneal benchmark in the PARSEC benchmark suite modifies a limited portion of the array per iteration

Because errors can be difficult or costly to detect, they are sometimes latent, and thus multiple versions can be used to improve overall performance and reliability [19]. This capability is beyond that of traditional checkpoint/restart systems that only maintain a single checkpoint; if there are latent errors that corrupt the checkpoint, there is no way to recover the system. Lu et al. show when multi-version checkpointing is useful [19] across a range of error and detection latency assumptions. The application-level abstraction of multi-version arrays creates a wide variety of opportunities for flexible error checking and recovery exploiting application semantics. However, those topics are the subject of other research studies.

B. Preserving Multiple Versions Efficiently

A central challenge for the multi-version foundation for resilience is how to implement versioning efficiently. The conventional method is to create an copy of the array for each new version, where denoted as the flat array approach. GVR limits modification to the current version of the array, limiting older versions to read-only which opens numerous avenues for optimization.

Our studies show that many applications modify only part of an array between versions. For example, Figure 2 shows the behavior of the canneal benchmark (from PARSEC [22]). We instrumented accesses to the main data structure called `netlist::_elements`, a contiguous array buffer, to understand modification patterns using the PIN tool [23]. This structure is the core needed for resilient execution. We assume that the array is divided into fixed-size blocks, and mark each block if the contents of the block is modified. Figure 2 shows that only a small amount of the array is updated during each iteration. Because the canneal benchmark runs for several iterations with a barrier synchronization at the end of each iteration, it naturally corresponds to a version. Our results show that a small fraction of the array is updated in each iteration, creating opportunity for optimization.

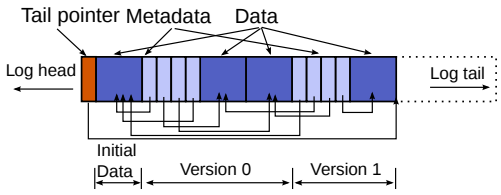


Fig. 3. In-memory data structure of log-structured array

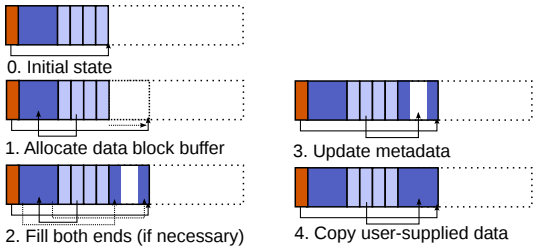


Fig. 4. Procedure for RMA Put Operation

III. Design

We present the design of log-structured implementations for global arrays. We first describe the in-memory data structure, then two implementations—RMA-based and message-based protocol.

A. Data Distribution

Each global array is a distributed collection of buffers that together comprise a single logical array. We assume that data distributions map each range of array indices to a corresponding remote memory buffer, and we assume the data distribution does not change across versions. For a given operation, the memory buffer (target) we need to access may be in a remote node. We use the term "client" to indicate the originating node and "server" for the target node.

B. Data structures

Figure 3 illustrates the in-memory log data structure of a log-structured array. An array consists of a single contiguous memory region, dividing it into two parts—data and metadata blocks. Within the array, a region of the global array is divided into fixed-size blocks, each storing a portion of user data. Each metadata block contains a pointer to a user data block. Thus for a given array size, we have a fixed number of metadata blocks for each version. For example, given that L is the length of array and B is block size, single version requires $\lceil L/B \rceil$ metadata blocks.

C. Operational semantics

There are two cases for a *put* operation. In the base case, new data blocks are allocated at the tail of the log to record the modified data. Then the corresponding metadata blocks are updated, pointing to the newly allocated blocks. If the put operation is overwriting data that has already been modified since the most recent version creation, then it simply overwrites the current data block. No new allocation is required. Thus new versions are created incrementally based on new modifications of a region.

Upon *version_inc()*, we can create a logical new version by simply creating a new set of metadata blocks for the version (similar to a copy-on-write process creation). The new metadata blocks are simply appended to the tail of the log. And the location of the metadata (current version), but not their contents is broadcast to all of the clients. At this moment, all metadata blocks are identical to those of the previous version.

If there are concurrent and non-conflicting *put* and *get* operations, the implementation must merge the updates and capture all modifications. GVR provides synchronization operations to order conflicting updates, and if operations are not well-ordered, then arbitrary interleavings of update are acceptable.

D. Data Access Protocols

A key feature of modern cluster networks is RDMA (Remote Direct Memory Access). RDMA can be high performance because it is 1-sided, not requiring involvement from the remote CPU. We present two access protocols for comparisons, one with RDMA and the other without RDMA. Hereafter we use more generic term RMA (Remote Memory Access), instead of RDMA.

1) RMA-based Protocol: Uses RMA operations only, with all data operations implemented by clients. The server exposes memory regions through RMA, but performs no operations.

a) Metadata cache: To access array data, a client needs the metadata blocks to find the location of the needed data blocks. Upon access, the client first checks the cache for the needed metadata, and if necessary fetches it from the remote node. Because the metadata may be correct even across a *version_inc()*, the metadata cache is not flushed at new version create. Instead, it is checked upon access, and if determined to be stale (failed access), then is it updated. As described in III-C, each metadata block is updated at most once in a single version. This means if a metadata block is already updated in the latest version, it will never change. Therefore, if a metadata cache is for the updated block, that cache is guaranteed to be always valid. As a result, each metadata cache has two states: *valid* and *maybe invalid*. Each client can determine the state of the cache without involving communications. Upon a version increment, all processes exchange the position of the log tail. If a metadata cache points to a location after the known log tail, that cache is valid because that data block is allocated in that version.

b) Put: RMA put requires a relatively complex procedure illustrated in Figure 4. Log area is exposed via the RMA interface as a single contiguous memory buffer. At a fixed location in the area, there is a special integer field to contain tail pointer of the log.

1) A client first tries to increment the tail pointer to allocate a new data block at the end of the log. This is done by an atomic operation.
2) If the user-supplied data to put is not block-size-aligned, one or two ends of the block need to be copied from the previous version.
3) The client updates metadata to point to the newly allocated data block. This update is also done by an atomic operation.

4) Finally, the client transfers user-supplied data to the data block.

If a client has valid metadata in the cache for its access, it means that the target data block is already allocated in the current version. Therefore, in such a case the client can skip steps 1 to 3, and start from the step 4. If the *put* targets a large contiguous area (*i.e.* the region spans multiple data blocks), steps 1, 2, and 4 can be done simultaneously. However, step 3 must be done one-by-one for each block because we assume that atomic operations only operate on a single integer.

c) Get: *get* is implemented in a speculative fashion. First, the client speculatively gets the target data based on its metadata cache entries. If the required metadata in the cache is valid, the get is complete. If the metadata is maybe-invalid, the client simultaneously fetches metadata from the server to update its metadata cache. If the fetched metadata matches the prior copies, then the fetched data is correct. If not, The data block must be reloaded based on the new metadata. If the get request is issued for large message size (*i.e.* message size that spans multiple data blocks), this speculative get is performed only if the data blocks pointed by the current metadata cache are contiguous, for the best performance.

d) Flushing Old Versions: Old version data should eventually be flushed out to secondary storage (*e.g.* NVRAM, SSD, HDD, or shared parallel file system). The log-structured array has a special mechanism to do this flushing for RMA-based access. When a client updates a remote metadata block, it records the location of the old memory block locally. When the number of old blocks reaches certain threshold, it sends the list of old blocks to the remote server thread. Then the server thread flushes these blocks to the secondary storage. With this mechanism, the main thread does not block when flushing the old blocks.

2) Message-based Protocol: This protocol is implemented by special messages that request array operations of a server. In order to handle messages, the server employs a dedicated thread for message handling. The server thread performs all the complex operations on the log structure, such as memory allocation and metadata updating. Then the server thread just returns a result (retrieved data for the get request, status code for the put request) to the client.

E. Design Considerations and Alternatives

1) Variable Block Size: As described above, log-structured arrays use fixed-size blocks. One alternative design choice would be to use variable-length block sizes. Variable block size would allow a single metadata block to represent a wider range, and would lead to reduce the size overhead of metadata blocks. However, this would make metadata addressing more difficult, especially for RMA clients. Therefore, we decided to use fixed-size block size.

2) Log Cleaning (Garbage Collection): The current design of the log-structured arrays does not include log cleaning. In order to make the system practical, a log cleaning feature is necessary when a log tail reaches the end of the memory buffer. One idea would be to run a garbage collector in *version_inc* call at each node. During version increment, it is guaranteed that no other process is going to modify the array.

F. Implementation

GVR is implemented on top of MPI-3[24], which offers advanced one-sided communication functionalities compared to MPI-2. In MPI, a program must create a *window*, which is an abstract object that exposes a particular memory region for remote access. A process can access a remote memory region exposed through a window using one-sided communication functions such as *MPI_Put* or *MPI_Get*. When a global array is created, GVR sets up an MPI window to expose the memory buffer of the array. Another important feature introduced in MPI-3 is a set of atomic operations. Particularly, GVR uses the *MPI_Compare_and_swap* function to implement an atomic update of data structures in log-structured array.

GVR also utilizes message-passing features in MPI. GVR launches one service thread for background communication. This thread works as a server to handle complex operation requests, such as array creation, array destruction, and version increment. It can also handle basic operations such as get/put of array data for message-based data access scheme. In order to maximize the throughput for accessing log-structured arrays via the message-passing style, the server thread uses a temporary, contiguous buffer for large message size operations. This buffering incurs one additional copy on the data movement path, but it helps reduce the total number of MPI communications, which determines the overall throughput.

IV. Evaluation

This section provides several benchmark results to reveal the performance of log-structured array, by comparing with flat array performance. First we exhibit a set of microbenchmarks, then demonstrate several performance data with scientific application programs.

A. Configuration

All the experiments are conducted on the Midway cluster installed in University of Chicago RCC. Each node has Intel Xeon E5-2670 (2.6GHz, 8-core), 32GB RAM, and Infiniband FDR-10 as an interconnect. One GVR process has two threads, the main thread and the target server thread (see Section III-F). We assign one dedicated CPU core for each thread.

1) Special Implementations for Midway: In order to achieve good performance on the Midway system, we had to add several machine-specific tuning to the GVR library. First, we found that if too many outstanding requests were issued on Infiniband, overall performance dramatically dropped in this environment. So we configured GVR so that it would call *MPI_Win_flush_all* for once in every 32 MPI one-sided communication functions to flush all outstanding operations. Second, we found that default implementation of *MPI_Waitany* of MVAPICH2 did not perform well in multi-thread programs. The server thread spends most of the time in *MPI_Waitany* to wait for an incoming message, but this heavily sacrifices the performance of the main (application) thread. To mitigate the issue, we employ a combination of *MPI_Testany* and a spin loop in the server thread to wait for a message.

2) GVR Array Configurations: We compare several different array implementations to reveal performance characteristics. Array configurations are shown in Table I. We introduce two axes, data structure on the memory and access method.

TABLE I. GVR ARRAY CONFIGURATIONS

	Remote Memory Access	Message-passing
Contiguous Buffer	Flat-RMA	Flat-msg
Log-structured Buffer	Log-RMA	Log-msg

TABLE II. BLOCK SIZE CONFIGS FOR LOG-STRUCTURED ARRAY

Experiment	Block Size (Bytes)
Microbenchmarks	128
Synthetic Workloads	128-8192 (varies)
OpenMC	8192
PCG	8192
canneal	512

Two data structures are used, *contiguous (flat) buffer* and *log-structured buffer*. The former is a simple, flat, and contiguous memory region, whereas the latter is a log-structured array proposed in this paper. Block size configurations of the log-structured array are shown in Table II. Block size is chosen based on the balance between runtime performance and array modification (update) ratio.

For each array data structure, two data access methods are provided and compared. One is *remote memory access (RMA)*, in which a client uses RMA to access a target buffer. The other is *message passing*, where a client sends a message to request a data access to a server thread on a remote server.

For the Log-RMA configuration, there are several sub-states regarding the state of metadata cache and target data block, as follows: *a) miss:* metadata cache misses in get operations. *b) hit:* metadata cache hits in get operations. *c) first:* the first put for the target data block in put operations, thus the operation should begin with allocating a new data block. *d) ow-miss:* the target data block is already allocated in put operations, but the client-side metadata cache misses. *e) ow-hit:* the target data block is already allocated in put operations and the client-side metadata cache hits.

3) Memory Configurations: We project that NVRAM will be used as a part of main memory in the future computer systems [25]. Such system will have a combination of DRAM and NVRAM as a main memory. At this moment we are not sure which memory technology will actually win, but in general NVRAMs are expected to have higher density, lower bandwidth, and in some cases lower write durability, compared to DRAMs.

NVRAM can be exploited as a storage for old versions, as they are read-only and rarely accessed. The log-structured array will match the NVRAM characteristics in several reasons. First, log-structured array can create versions incrementally, so it can flush memory blocks to NVRAM in background. Second, it reduces the size of the versions preserved in NVRAM, which increases the number of versions kept or brings longer lifetime.

We add "slow" version of memory copy function to simulates flushing out of old version from DRAM to NVRAM, using the `rdtscp` instruction loop as in [26] We assume that the bandwidth of NVRAM is 1/10 of DRAM [25], so we tuned the slower memory copy to take 10x latency compared to regular one. We also prepared 1/100 speed configuration to simulate SSD. We assume only old version data goes to slower NVRAM or SSD, and everything else, including current version data and internal data structures of the GVR library, is located on faster DRAM. When creating a version, flat array needs to wait for memory copy to complete, but log-structured array does not have to block as it has a background flushing mechanism (see Section III-D1d).

B. Micro-benchmarks

1) Latency: Access latency for individual operation is measured. For get, latency means the time taken between the operation being issued and the value being ready in a local buffer. For put, this means the time taken for the data being written to the remote node.

Figure 5 shows the latency results. For get operations, Log-RMA requires higher overhead than the Flat-RMA when the metadata cache misses. This is because one additional round-trip communication is required to retrieve the latest metadata information to access the actual data. If the cache hits, the latency is almost the same as Flat-RMA, as direct access to the data block is possible in this case. For the first put operations in Log-RMA, the latency gets higher if the message size gets bigger than 128 ($= 2^7$) bytes, which is a block size for this experiment. This is because first put requires metadata updates and these updates require atomic compare-and-swap operations for each individual metadata block. This increases the number of one-sided MPI operations and limits the performance. The *ow-miss* (overwriting, metadata cache miss) cases in Log-RMA show the similar performance as the *first* cases for large message size. This may be improved because overwriting does not require metadata updates. Both Flat-msg and Log-msg configuration had similar high latency compared to RMA-based schemes. This is because in message-based schemes the server side thread has to be scheduled and receive the request in order to handle it.

2) Bandwidth: Figure 6 shows single client, single server bandwidth. For get operations, Log-RMA configuration achieves almost comparable (at least 74%) performance to the Flat-RMA configuration, if the metadata cache hits. If the cache misses, it becomes around 50% relative to Flat-RMA. This is reasonable because one additional metadata fetch is required in the cache miss case. For put operations, in the best case Log-RMA achieves 93% of bandwidth compared to Flat-RMA at 4-byte message size, however we observe that the performance of the first put cases in Log-RMA saturates after the message size goes beyond $2^7 = 128$ bytes. In the worst case, for 8192-byte message size, Log-RMA performance becomes 1% of the Flat-RMA As described before in Section IV-B1, this is due to atomic operations for metadata updates. Message-based schemes perform pretty bad compared to RMA-based schemes, in almost all message sizes. Therefore we further focus on RMA scheme only.

According to the observations above, log-structured array can be competitive with flat array, if RMA-based access is performed and metadata cache hits. However the overhead for put operation is large if a new block allocation is required (*i.e.* "first" case). These results suggest that log-structured array is more suitable for read-dominant workloads in terms of best runtime performance.

3) Version Increment Cost: Cost for version increment operation is measured for both flat array and log-structured array. As described in Section III-F, the version increment

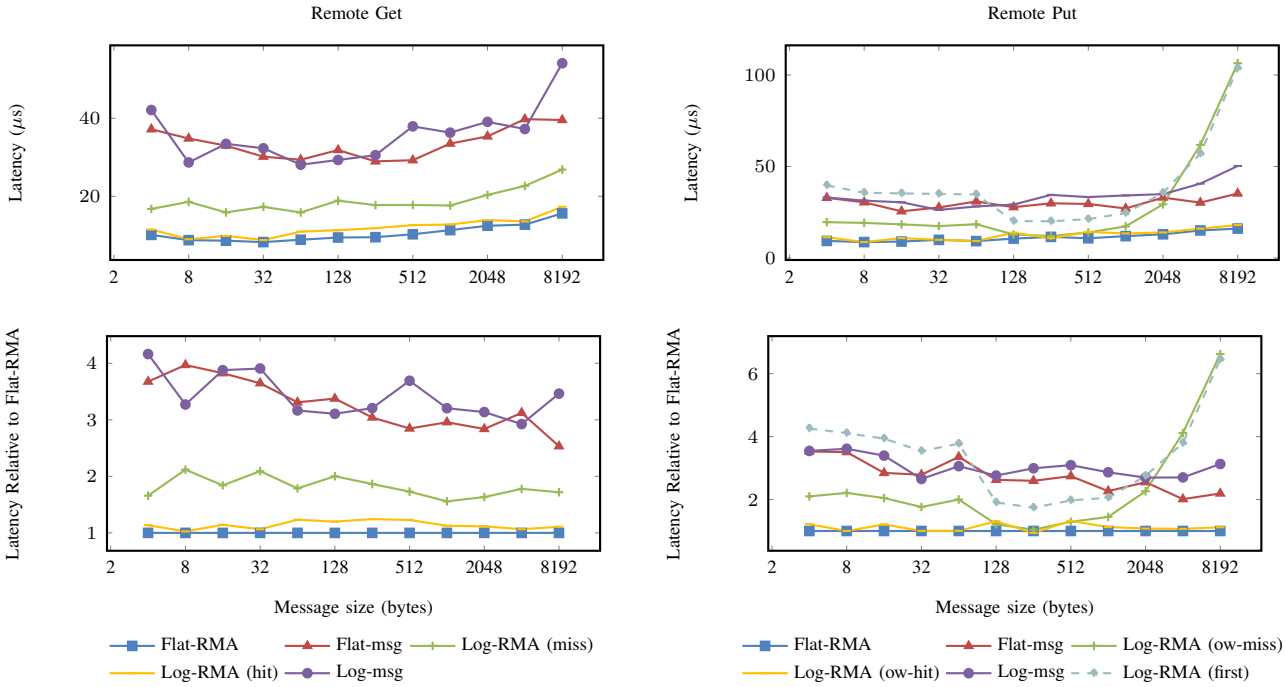


Fig. 5. Access Latency for Put and Get operations (varied implementations)

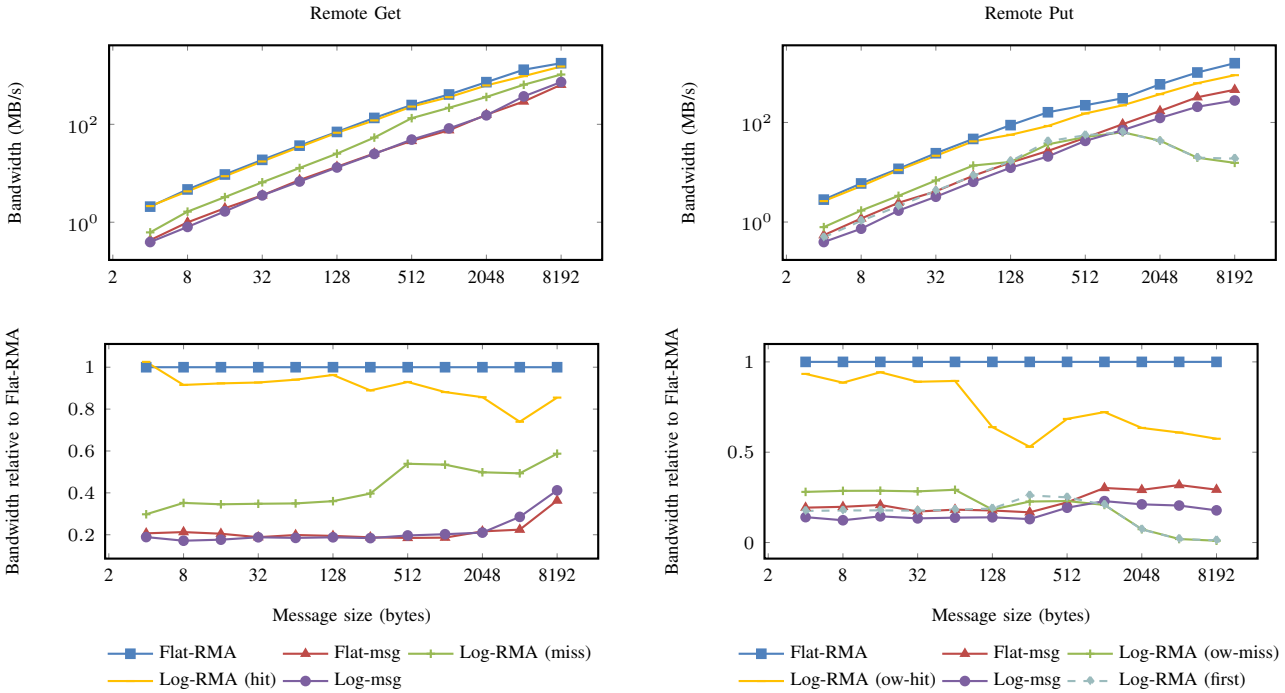


Fig. 6. Single client, Single server Bandwidth, various configurations

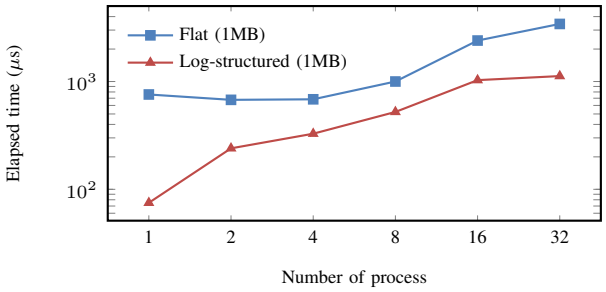


Fig. 7. Elapsed time for version_inc() call, 1MB array

```
while (true) {
  for (i < n_reads)
    { loc=rndloc(); get(loc, gds); }
  wait(gds); /* Wait for outstanding operations
      to complete */
  for (i < n_writes)
    { loc=rndloc(); put(loc, gds); }
  version_inc(gds);
}
```

Fig. 8. Pseudo-code of the kernel of the synthetic workload.

is done on a server-side thread, so this cost is independent from data access schemes. It only depends on the array data representation.

This benchmark first creates an array with a size of 1MB/process, then fills data to the entire array. After that it calls *version_inc* function to increment version. Finally, time to complete the *version_inc* call is measured. This experiment is done on 1 node through 32 nodes.

Figure 7 shows the result. Log-structured array shows significantly low cost compared to flat array, as much as 10 times speed up in single process case. This is because the flat array has to copy the entire array upon version increment but log-structured array only needs to copy metadata blocks, which are just 3.125% of the array in this case (metadata size = 4bytes, block size=128 bytes).

C. Synthetic Workloads

Next we show the results of synthetic workloads shown in Figure 8 to compare the performance characteristics of log and flat array. The benchmark first allocates one-dimensional global array with uniform data distribution (*i.e.* each process owns the same size of buffer), then each process repeatedly reads and writes to random locations, and creates a new version at a certain interval (every `n_reads + n_writes` operations). A location is generated by the following function, similar one used in APEX-Map [27]:

$$rndloc = C_i + s\frac{L}{2}p^{1/k}$$

where L stands for the total length of the array, n for total number of processes, i is the rank of each process ($0 \leq i < n$), and $C_i = \frac{L}{n}i + \frac{L}{2n}$ which is the center of the memory buffer owned by process i (assuming that process i owns $i+1$-th memory chunk of the array). s and p are random variables, s gives either 1 or -1 in the same probability, whereas p gives a uniformly distributed random values over $[0, 1]$. $k (0 < k \leq 1)$ determines the shape of the output of this function. Essentially is represents the locality of the access. If k equals to 1 it shows no locality, whereas when k becomes smaller and gets closer to 0, memory accesses will show higher locality around C_i. We chose three values for k, $0.0025, 0.025, 0.25$, each of which corresponds to memory access patterns of radix sort, N-body simulation, and matrix multiplication respectively[27]. For each location chosen, 128 bytes of data are accessed in single operation.

TABLE III. APPLICATION CHARACTERISTICS

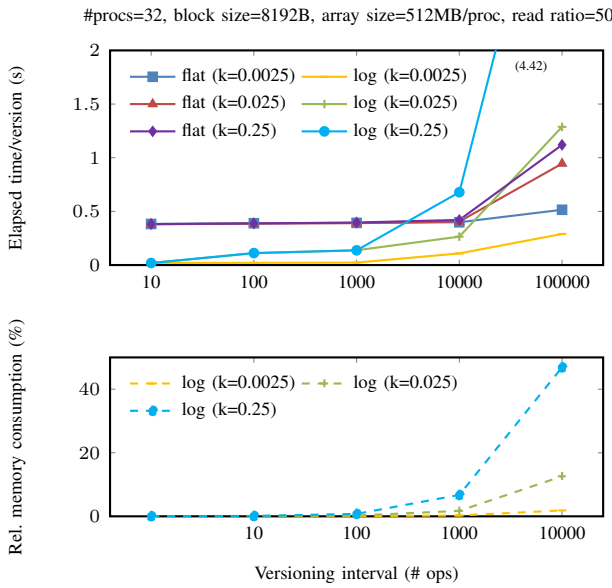
	OpenMC	PCG	canneal
Access type	Accumulate only	Put only	Put and Get
Access pattern	Random	Regular	Random
Access size	Small (8 bytes)	Large (entire array)	Small (8 bytes)
Scaling	Strong for array, Weak for particles	Strong	Weak
# of GVR arrays	1	3	Strong / Weak
Size of each array	67MB	8MB	N_{proc}
Versions captured	10	114 - 141	36MB / 10

Results are shown in Figure 9. Figure 9(a) (upper graph) indicates that the log-structured array outperforms flat array when versioning interval is small (=high frequency), due to smaller *version_inc* cost. In particular, at the most frequent 10 ops/version (5 puts/version) case, log array is more than 19x faster than flat array. At this point memory consumption of the log array is less than 0.06%, saving more than 99%. If versioning interval becomes greater, *version_inc* cost for flat array gets amortized and achieves higher performance than the log array. Large k also makes log array performance worse because locality becomes smaller. The bottom graph in Figure 9(a) shows the memory consumption relative to flat array. It shows that larger k or longer interval makes the program update more blocks, thus it consumes more memory. Figure 9(b) shows how the performance is affected by different process/array sizes. By scaling up the number of processes, log array performance gets worse because it will increase the contention between processes. However on the other hand log array is not affected so much by array size scaling. Figure 9(c) shows that higher read ratio benefits log array, as the put may require block allocation and costly. This confirms that log array suits well for read-intensive workload. Finally, Figure 9(d) illustrates the tradeoff in log array between performance and memory savings. Large block size improves the performance because it raises the chance of the block being reused (thus lowering block allocation cost), but if there is less reuse the most of the allocated block will become a waste in terms of memory consumption.

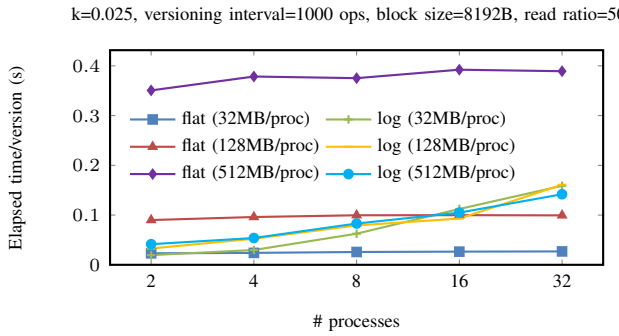
D. Application-level Benchmark

Here we present more realistic evaluations using three different scientific applications: OpenMC, PCG, and canneal. Table III summarizes the characteristics of three applications.

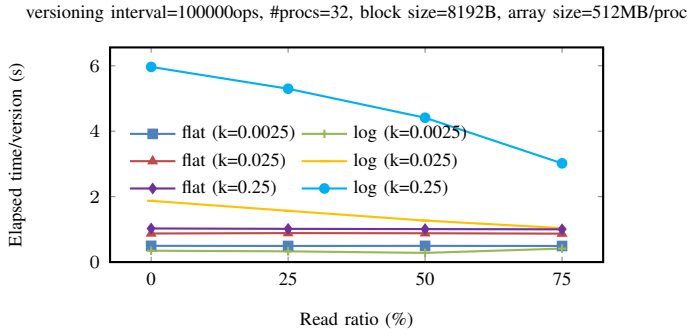
1) OpenMC: OpenMC is a production Monte Carlo code [28] for nuclear reactor simulations. We integrated GVR into OpenMC by applying GVR to its tally data. Tally is region-based and accumulated (*i.e.*, fetch-and-add) data, where the region, or tally region, is the volume over which the tallies should be integrated. In a realistic reactor simulation, tally could reach terabytes size of data. GVR enables tally decomposition



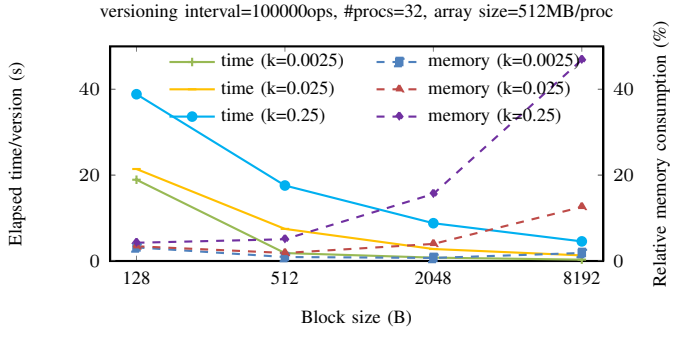
(a) Performance/memory consumption with various versioning intervals



(b) Performance with process/array size scaling



(c) Performance with varied read/write ratio



(d) Performance/memory saving tradeoff with various block sizes

Fig. 9. Results of the synthetic workload

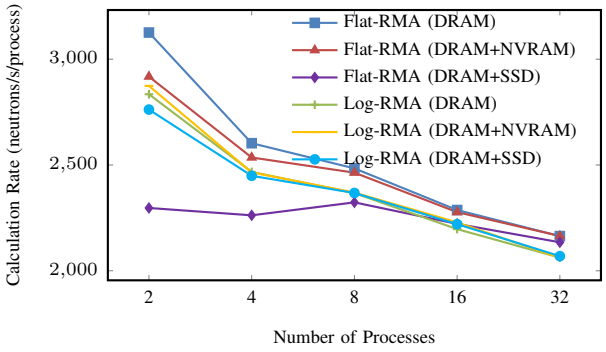


Fig. 10. OpenMC Performance (computation rate), with several memory configurations

and introduces resilience by versioning at the end of each batch, where batch is the grouping of multiple realization (particle histories) for tally purpose. Integrating GVR arrays requires fairly small changes (less than 1%) of OpenMC code. During the simulation, the tally is directly accumulated to the GVR global array. When a batch of particles simulation finished, a version is created by *version_inc()* We configured the total tally size to 67MB, and the number of neutrons to 5000/process. Therefore the simulation is weak scaling in terms of number of neutrons and strong scaling in terms of the size of tally.

Figure 10 shows overall results of OpenMC. Since the tally size per process shrinks as the number of processes increases, results are plotted in per-process performance. When we look at the DRAM results, Log-RMA performs comparable to Flat-RMA. For 32 nodes, Log-RMA is only 4.7% slower compared to Flat-RMA. While achieving similar performance, log-structured array consumed 14.5% less memory to preserve versions, as shown later in Figure 13. Figure 10 also compares performances when NVRAM or SSD is introduced. Performance difference is most significant in 2-process case where each process holds the biggest size of data. In 2-process case, Flat-RMA performance is significantly dropped when NVRAM or SSD is introduced in the system. However for Log-RMA, NVRAM or SSD adds smaller impact to the performance. In the most extreme case for 2 processes, where SSD is introduced, Log-RMA outperforms Flat-RMA by 20%. This is because Flat-RMA is blocked at slow memory copy at each version increment while Log-RMA is not.

2) PCG Solver: Preconditioned Conjugate Gradient method (PCG) is a common way to solve linear systems (*i.e.* find x in $Ax = b$) [29]. Our implementation uses the linear algebra primitives Trilinos library [30]. The vectors used in the computation are stored in a customized variant of a Trilinos Vector class that supports preservation and restoration of values via a GDS object, and versioned every iteration. Total number of versions (= number of iterations) depends on the number of processes, ranging from 114 (for 2 processes) to 141 (for 32 processes). For this study, we use for A a sparse matrix derived from the HPCG benchmark [31] of size 1000000×1000000.

Figure 11 shows the results of the PCG solver experiment. This program shows a quite unstable behavior when the number of processes becomes more than eight, so we pick the most stable run among three trials. The Log-RMA result is pretty close to Flat-RMA performance, even in the worst case the

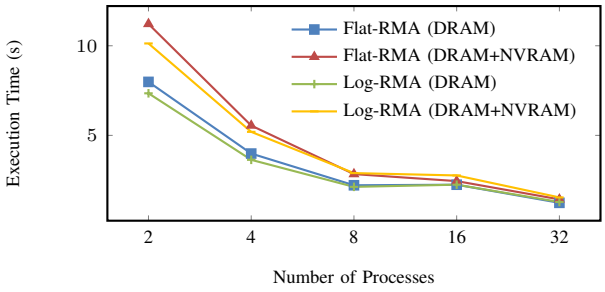


Fig. 11. Preconditioned conjugate gradient (PCG) solver runtime with NVRAM emulation (seconds)

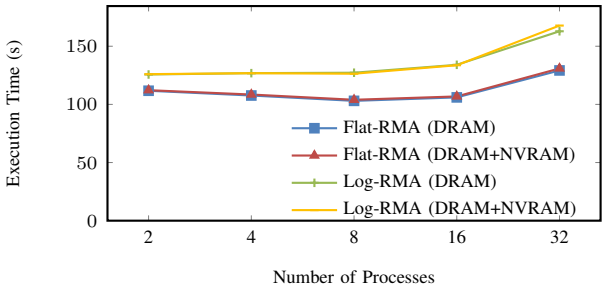


Fig. 12. canneal replay runtime with various memory configurations (seconds)

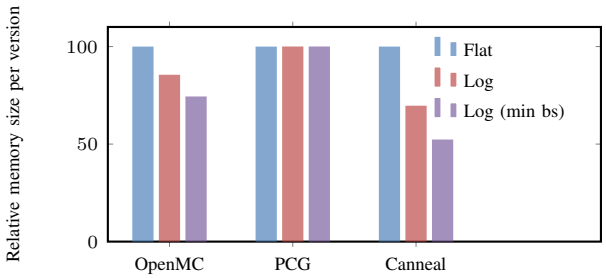


Fig. 13. Version memory usage (flat array = 100)

additional overhead is just 3.7%. This program creates versions more than 100 times during the run, the versioning cost is important. As shown in Figure 11, putting slower NVRAM into the system heavily affects the performance. In this experiment even Log-RMA is affected by NVRAM, possibly because versioning frequency is too high. For this application, there is no memory savings by Log-structured array because it overwrites the entire region for every version.

3) canneal: Third application benchmark is a synthesis benchmark based on canneal from the PARSEC benchmark suite. Canneal is a multi-threaded program which simulates an optimization process of an electric circuit. It has a shared array which stores a huge list of elements of a circuit. The program tries to swap two randomly-chosen elements and if the swap improves the circuit, then the result of swapping is written back to the array. The goal of this benchmark is to reproduce the same access pattern to the array using GVR.

We developed a trace-replay framework for doing this. First, memory access trace of the target program is recorded using PIN tool[23]. Then we replay the trace by a GVR program that translates the memory access log to GVR calls. Specifically, malloc() is replayed as *alloc()* in GVR, memory read and write are as *get()* and *put()*, separately, and synchronization points are translated as *version_inc()*. *get()* is followed by the *wait_local()* call to make sure that the result of *get()* is available, because in GVR *get()* is implemented as a nonblocking function. To make the workload more realistic, $10\mu s$ of dummy computation is inserted after each *get()*. This is almost the same latency for one remote memory access. Additionally, to make the replay run weak-scaling, we replicate the orignal array multiple times, because the orignal program is strong-scaling. For n-process run, the array is replicated so that all processes share n arrays. Then each process replays the same trace for each array. In this way, the array size as well as the amount of workload scales as number of processes grows.

We generated the trace using the "simlarge" input file. Then we replayed this trace on GVR array, using up to 32 compute nodes, 1 process per each node. In the original program, it requires 128 synchronizations (= *version_inc()* in replay), but to reduce the runtime of the replay, we just replayed until first 10 synchronizations.

Figure 12 shows the result of replay time. Log-RMA has at most 26% percent overhead compared to Flat-RMA. Given that this workload is quite communication intensive, this could be a sort of worst case scenario for the log-structured array. If the amount of computation increases, this overhead will be amortized at certain point. While the log-structured array has an overhead for canneal, it saves significant amount of memory for preserving versions. As shown in Figure 13, it saves about 30% of memory compared to flat array versioning.

4) Version Memory Usage: Figure 13 compares the memory size required to preserve version data of each application benchmark. Size is shown relative to flat array. *Log* shows the memory usage when the block size shown in Table II is applied, while *Log (min bs)* shows when the block size is equal to the message size that the program issues (*i.e.* 8 bytes for OpenMC and canneal, whole array size for PCG). So this is the actual size of the regions that is modified by the program. These results are measured in 32-node run. For canneal, the maximum memory saving would be as much as 47.7

V. Related Work

The design of the log-structured array is strongly motivated by the log-structured file system[32]. While the log-structured file system is designed for a file system on top of classical rotating hard disk drives, our log-structured array is designed for multi-version memory. Our Log-structured arrays have several characteristics optimized for efficient remote memory access, such as pre-allocated, and fixed-size metadata (index) structures. Behavior on overwriting is also different. The log-structured file system keeps all the updates in the log, while the log-structured array keeps only the latest update in a particular version.

Similar structures or designs have appeared in several distributed key-value store systems. Pilaf[33] is a key-value store that utilizes RDMA operations, however it applies RDMA only for get (read) operation from a server. Put (write) operation is always handled by message-based scheme. Our log-structured array implements both RMA-based and message-based access methods for both get and put, and shows empirical performance results. RAMCloud[34] is a distributed key-value store designed

for fast crash recovery. In order to deal with a crash, it dumps newly appended data to disk storage. Its internal data organization is a log-structured memory. It also implements log cleaning schemes [35], a part of which could be utilized in our log-structured memory for GVR. SILT[36] is a distributed key-value store system that combines DRAM and flash . It stores all the appended data in a log structure, and older logs are flushed to a flash device with compression. Both in RAMCloud and SILT, written logs are considered read-only. This allows the systems to efficiently handle the log, for example in applying compression. Since the older version data in GVR arrays are also read-only, GVR will be able to apply similar techniques to old versions.

VI. Summary and Future Work

We presented a log-structured array data structure for efficient memory versioning, and two different underlying communication models— RMA and message-passing. Our results show that log-structured arrays using the RMA access scheme can be comparable to flat arrays using the RMA scheme in terms of access performance, and much faster in version creation. Application-level studies show that log-structured versioning can achieve good access performance and fast version creation in full applications, delivering good efficiency overall. Further, the memory savings delivered by log-based schemes can be dramatic.

Interesting future directions include exploration of additional versioning implementations that might exploit application or hardware assistance (*e.g.* dirty bits), additional implementation issues such as log cleaning, and broader variety of hardware platforms. Other directions include use in systems that persist many versions, where the problem of minimizing version storage size and tolerating failures naturally leads to explorations of compression, efficient redundancy, recovery, etc. Finally, broader evaluation using more applications is always valuable.

Acknowledgments

We thank Kamil Iskra, Ziming Zheng, and Aiman Fang for their helpful comments on the application study and performance evaluations. We also thak Amirali Shambayati, Pavan Balaji, and Antonio J. Peña for giving us a discussion on future memory systems. This work was supported by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Award DE-SC0008603 and Contract DE-AC02-06CH11357 and completed in part with resources provided by the University of Chicago Research Computing Center.

References

[1] Peter Kogge, et. al., "Exascale computing study: Technology challenges in achieving exascale systems," DARPA IPTO Study Report, available from http://users.ece.gatech.edu/mrichard/ExascaleComputingStudyReports/exascale_final_report_100208.pdf, 2008.

[2] S. Borkar and A. A. Chien, "The future of microprocessors," *Communications of the ACM*, vol. 54, pp. 67–77, May 2011.

[3] M. Elnozahy, "System resilience at extreme scale: A white paper," 2009, dARPA Resilience Report for ITO, William Harrod.

[4] F. Cappello, A. Geist, W. Gropp, L. Kale, W. Kramer, and M. Snir, "Towards exascale resilience," *International Journal of High Performance Computing Applications*, vol. 23, no. 4, pp. 374–388, 2009.

[5] B. Schroeder and G. A. Gibson, "A large-scale study of failures in high-performance computing systems," in *Proceedings of DSN*, 2006.

[6] Z. Zheng, L. Yu, W. Tang, Z. Lan, R. Gupta, N. Desai, S. Coghlan, and D. Buettner, "Co-analysis of RAS log and job log on Blue Gene/P," in *Proceedings of IPDPS*, 2011.

[7] K. Ferreira and et al., "Evaluating the viability of process replication reliability for exascale systems," in *Proceedings of Supercomputing*, 2011.

[8] K. Bergman and et al., "Exascale computing study: Technology challenges in achieving exascale systems," *DARPA IPTO Tech. Rep*, 2008.

[9] A. Moody, G. Bronevetsky, K. Mohror, and B. R. De Supinski, "Design, modeling, and evaluation of a scalable multi-level checkpointing system," in *Proceedings of Supercomputing*, 2010.

[10] "Workshop on silicon erros in logic systems," 2013.

[11] D. Fiala and et al., "Detection and correction of silent data corruption for large-scale high-performance computing," in *Proceedings of Supercomputing*, 2012, p. 78.

[12] C.-d. Lu and D. A. Reed, "Assessing fault sensitivity in MPI applications," in *Proceedings of Supercomputing*, 2004.

[13] M. Shantharam, S. Srinivasmurthy, and P. Raghavan, "Characterizing the impact of soft errors on iterative methods in scientific computing," in *Proceedings of Supercomputing*, 2011.

[14] A. A. Hwang, I. A. Stefanovici, and B. Schroeder, "Cosmic rays don't strike twice: understanding the nature of DRAM errors and the implications for system design," in *Proceedings of ASPLOS*, 2012.

[15] J. Lidman, D. J. Quinlan, C. Liao, and S. A. McKee, "ROSE::FTTransform-a source-to-source translation framework for exascale fault-tolerance research," in *Proc. of DSN-W*, 2012.

[16] G. Bronevetsky and B. de Supinski, "Soft error vulnerability of iterative linear algebra methods," in *Proceedings of ICS*, 2008.

[17] M. Hoemmen and M. A. Heroux, "Fault-tolerant iterative methods via selective reliability," in *Proceedings of Supercomputing*, 2011.

[18] S. K. S. Hari, S. V. Adve, and H. Naeimi, "Low-cost program-level detectors for reducing silent data corruptions," in *Proceedings of IPDPS*, 2012.

[19] G. Lu, Z. Zheng, and A. A. Chien, "When is multi-version checkpointing needed?" in *Proceedings of the 3rd Workshop on Fault-tolerance for HPC at Extreme Scale*, ser. FTXS '13. New York, NY, USA: ACM, 2013, pp. 49–56.

[20] G. Aupy, A. Benoit, T. Herault, Y. Robert, F. Vivien, and D. Zaidouni, "On the combination of silent error detection and checkpointing," in *Dependable Computing (PRDC), 2013 IEEE 19th Pacific Rim International Symposium on*, Dec 2013, pp. 11–20.

[21] J. Nieplocha, B. Palmer, V. Tipparaju, M. Krishnan, H. Trease, and E. Apr, "Advances, applications and performance of the Global Arrays shared memory programming toolkit," *International Journal of High Performance Computing Applications*, vol. 20, no. 2, pp. 203–231, Summer 2006.

[22] C. Bienia, "Benchmarking modern multiprocessors," Ph.D. dissertation, Princeton University, January 2011.

[23] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '05. New York, NY, USA: ACM, 2005, pp. 190–200.

[24] Message Passing Interface Forum, http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf.

[25] A. Huffman and D. Juenemann, "The nonvolatile memory transformation of client storage," *Computer*, vol. 46, no. 8, pp. 38–44, 2013.

[26] S. Pelley, T. F. Wenisch, B. T. Gold, and B. Bridge, "Storage management in the NVRAM era," *PVLDB*, vol. 7, no. 2, pp. 121–132, 2013.

[27] E. Strohmaier and H. Shan, "Architecture independent performance characterization and benchmarking for scientific applications," in *Modeling, Analysis, and Simulation of Computer and Telecommunications Systems, 2004. (MASCOTS 2004). Proceedings. The IEEE Computer Society's 12th Annual International Symposium on*, Oct 2004, pp. 467–474.

[28] P. K. Romano and B. Forget, "The OpenMC Monte Carlo particle transport code," *Ann. Nucl. Energy*, vol. 51, pp. 274–281, 2013.

[29] Y. Saad, *Iterative methods for sparse linear systems*. Society for Industrial and Applied Mathematics, 2003.

[30] M. A. Heroux, R. A. Bartlett, V. E. Howle, R. J. Hoekstra, J. J. Hu, T. G. Kolda, R. B. Lehoucq, K. R. Long, R. P. Pawlowski, E. T. Phipps *et al.*, "An overview of the Trilinos project," *ACM Transactions on Mathematical Software (TOMS)*, vol. 31, no. 3, pp. 397–423, 2005.

[31] J. Dongarra and M. A. Heroux, "Toward a new metric for ranking high performance computing systems," *Sandia Report, SAND2013-4744*, vol. 312, 2013.

[32] M. Rosenblum and J. K. Ousterhout, "The design and implementation of a log-structured file system," *ACM Trans. Comput. Syst.*, vol. 10, no. 1, pp. 26–52, Feb. 1992.

[33] C. Mitchell, Y. Geng, and J. Li, "Using one-sided RDMA reads to build a fast, cpu-efficient key-value store," in *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, ser. USENIX ATC'13. Berkeley, CA, USA: USENIX Association, 2013, pp. 103–114.

[34] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum, "Fast crash recovery in ramcloud," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, ser. SOSP '11. New York, NY, USA: ACM, 2011, pp. 29–41.

[35] S. M. Rumble, A. Kejriwal, and J. Ousterhout, "Log-structured memory for DRAM-based storage," in *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST 14)*. Santa Clara, CA: USENIX, 2014, pp. 1–16.

[36] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky, "SILT: a memory-efficient, high-performance key-value store," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, ser. SOSP '11. New York, NY, USA: ACM, 2011, pp. 1–13.