Empirical Comparison of Three Versioning Architectures

Hajime Fujita,*† Kamil Iskra,† Pavan Balaji,† Andrew A. Chien*†
*Department of Computer Science
University of Chicago
1100 East 58th Street, Chicago, IL 60637, USA
†Mathematics and Computer Science Division
Argonne National Laboratory
9700 South Cass Avenue, Argonne, IL 60439, USA
hfujita@cs.uchicago.edu, iskra@mcs.anl.gov, balaji@anl.gov, achien@cs.uchicago.edu

***Abstract*—Future supercomputer systems will face serious reliability challenges. Among failure scenarios, latent errors are some of the most serious and concerning. Preserving multiple versions of critical data is a promising approach to deal with such errors. We are developing the Global View Resilience (GVR) library, with multi-version global arrays as one of the key features. This paper presents three array versioning architectures: flat array, flat array with change tracking, and log-structured array. We use a synthetic workload comparing the three array architectures in terms of runtime performance and memory requirements. The experiments show that the flat array with change tracking is the best architecture in terms of runtime performance, for versioning frequencies of 10^{-5} $\mathbf{ops}^{-1}$ or higher matching the second best architecture or beating it by over 8 times, whereas the log-structured array is preferable for low memory usage, since it saves up to 88% of memory compared with a flat array.**

I. Introduction

As supercomputer systems evolve toward exascale systems, several challenges are anticipated. High failure rate is one such challenge [1], due to several technology trends such as increasing scale, shrinking process size, and low power voltage [2]. Failures are already an issue even in today's large-scale systems [3]. Among various modes of failures, *latent errors* (or silent data corruption, SDC), which corrupt data in a way that cannot be detected immediately, are becoming a serious concern [4].

To address these issues, we have been developing the Global View Resilience (GVR) library [5]. GVR is a lightweight library that adds flexible application-level resilience into large-scale scientific applications. It has two key features: multi-version, multi-stream distributed arrays and a unified error handling interface that supports flexible cross-layer error checking and recovery. Multi-versioning is a promising approach for handling latent errors [6], since a high probability exists that some versions have been created before the latent error corrupted the data. Introducing the concept of multi-version arrays, however, immediately raises a question of the cost of creating and keeping such multiple versions. In previous work we proposed the log-structured array [7], which records updated data blocks into a log, for efficient versioning. However, that work did not explore other possibilities for versioning architectures or the cost of restoring data from an old version.

In this paper, we compare three array versioning architectures:

- *Flat array:* Uses a simple contiguous array representation, with full copy taken on each version creation
- *Flat array with change tracking:* Uses the same flat array for the most recent version but preserves only the modified data blocks upon version creation
- *Log-structured array:* Keeps only the updated data blocks but does not hold a flat array

We address the following research questions: *Which array architecture brings the best performance and lowest memory consumption, under various workload characteristics? What are the trade-offs? What are the criteria for choosing different array architectures?* To illustrate the differences among these array architectures, we conduct a set of empirical benchmark tests using a synthetic application.

The overall conclusion is that log-structured arrays are preferable in terms of memory consumption, whereas the flat with change tracking architecture becomes preferable at moderate or low versioning frequencies.

II. Multi-Versioning in Global View Resilience

The Global View Resilience library enables resilient execution of large-scale scientific applications on unreliable hardware, by providing application-controlled, portable, and flexible error handling.

One of the key features of GVR is multi-version, multi-stream global arrays. GVR provides PGAS-style distributed arrays where applications can store their critical data and restore it later in case of errors. Accesses to arrays are explicit in most cases; applications invoke *put* or *get* library calls when accessing the data. While its basic interface is based on Global Arrays [8], GVR also provides a novel capability of preserving *multiple versions* of the array contents. The time of the versioning is controlled completely by the application. Creating a version involves an explicit *version_inc* call.

Past versions are read only, enabling, for example, transferring them to a remote process or other storage such as local SSD and shared parallel file system or applying error correction codes, compression, or encryption. Since applications work mainly on the current version, the runtime performance over the current version is an important metric. Applications can navigate through the version history and

retrieve data from an arbitrary version using the *get* function. An application can retrieve only a part of the old version, returned to a user-supplied separate buffer region.

GVR is implemented as a user-level library built on top of the MPI-3 [9]. GVR extensively utilizes MPI-3's one-sided communication (i.e., remote memory access, RMA) feature for basic data access operations such as put and get.

Multiple versions can be useful during recovery from latent errors [6]. Traditional checkpoint/restart systems keep only the latest checkpoint, because they assume that checkpoint data is correct. With latent errors, however, this assumption no longer holds. If the application discovers the error after a checkpoint has been taken, the only way to recover is by restarting the whole computation from the beginning. Having multiple versions addresses this issue.

The simplest way to implement a multi-version array is to make a full copy of an array upon version creation. However, our studies found that some applications do not modify the whole array region. For such applications, making a full copy each time makes little sense. Capturing only the modified region would save a lot of memory, at the same time enabling faster version creation by reducing the amount of memory to copy. In previous work we proposed the log-structured array to exploit this opportunity [7]. In this paper we propose another architecture that combines flat array and change tracking.

III. Design

In this section we describe several array versioning architectures that we are going to compare.

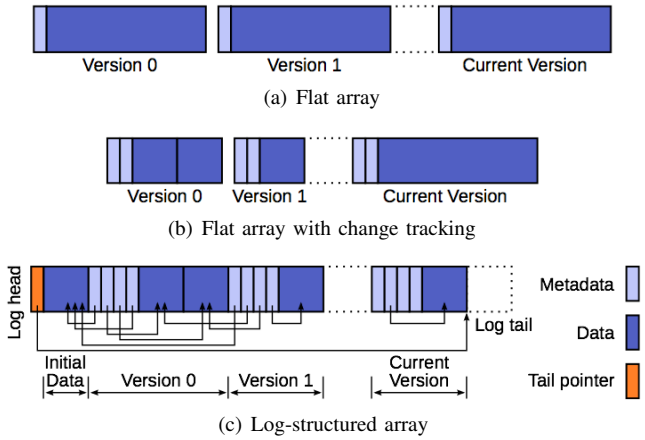
A. Flat Array

The array is represented by using a flat, contiguous memory buffer (Figure 1(a)). This buffer represents the current (newest) version. Put and get are performed directly on this region by using one-sided communication functions. When creating a version, a full copy of the entire region is made and kept for future reference.

B. Flat Array with Change Tracking

To reduce the overheads of flat array architecture, we implemented several change-tracking schemes within the local resilient data store (LRDS) component of GVR.

The first scheme, called "User," involves tracking changed memory areas by using a bitmap that is updated based on explicit information provided separately by the caller. On version creation, only the regions marked as modified are copied, creating an incremental version and thus reducing the overheads (Figure 1(b)). This scheme depends on keeping the change-tracking bitmap up to date, a task that could be burdensome and could introduce overheads that accumulate with every memory access. The multi-version array abstraction of GVR eliminates part of the burden: each GVR put operation is followed by an accumulate that sets appropriate dirty bits in a bitmap stored on the server. This accumulation is also implemented by using MPI one-sided communication.



(a) Flat array

(b) Flat array with change tracking

(c) Log-structured array

Figure 1. Data structures of each array versioning architecture

An improved tracking scheme, called "Kernel," takes advantage of OS kernel-level, page-based memory protection. On version creation, the memory buffer representing the current version is write protected, resulting in subsequent page faults on first write accesses to each page. A custom signal handler marks the faulting page in the bitmap as changed and unprotects it. The chief advantage of this scheme is that it is transparent to regular memory accesses from user code; unfortunately, it is not transparent to the system software. While the scheme works with InfiniBand, communication falls back to utilizing an intermediate buffer behind the scenes, resulting in a latency increase by an integer factor.

C. Log-Structured Array

The log-structured array architecture was proposed in previous work [7]. Unlike the two architectures discussed above, this one does not use a contiguous buffer to represent the current version. Instead, a memory buffer is allocated on demand when the first write (put) to a particular region takes place. These dynamically allocated data blocks form a log, so we call the whole architecture the log-structured array (Figure 1(c)). Each metadata block corresponds to a particular array index range and points to the corresponding data block. Each version has one set of metadata blocks, allowing one array to have multiple versions.

Put and get operations are implemented by using MPI one-sided communication. Since the data blocks are indirectly pointed to by metadata blocks, each data access requires at least two round trips, one for retrieving metadata and the other for actual data access (get/put). To reduce the latency, the log-structured array caches metadata on the client side.

IV. Evaluation

In this section we show empirical comparisons of the three versioning architectures. Evaluations are done based on two criteria: runtime performance in failure-free run and memory consumption.

A. Workload

We used a synthetic workload capable of generating various array access patterns, in terms of versioning frequency, read/write (get/put) ratio, access locality, and so on. This program continuously writes to/reads from an array, with an access width of 64 bytes. The target location is randomly determined by the following formula, based on APEX-Map [10]:

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

where C_i stands for the center coordinate of the array index range owned by process i, s is a random sign variable that becomes either 1 or -1, L is the total size of the array, p $(0 \leq p \leq 1)$ is a random variable with uniform distribution, and k $(0 < k \leq 1)$ is a parameter to control the spatial locality of the access sequence. In short, each process makes a large number of array accesses, clustered around the memory region owned by the process. We show results for $k = 0.025$, which corresponds to the memory access pattern of an N-body simulation [10], since we found it to be a good representative of the overall trends.

B. Setup

We used the Midway cluster installed at the University of Chicago Research Computing Center. Each node has two Intel Xeon E5-2670 (2.6 GHz, 8-core) CPUs, 32 GiB RAM, and InfiniBand FDR-10 as an interconnect. MVAPICH2 2.1rc1 was used as an MPI library. When multiple processes were used, we assigned four processes per node.

The array configurations were as follows:

- *flat*: flat array
- *flat-{user,kernel}*: flat with change tracking array using either the user or kernel scheme
- *log*: log-structured array

For simplicity, we stored all versions in memory.

C. Performance Comparison

We compared the performance of the synthetic workload, measured as total throughput; the results are in Figure 2. This benchmark suffers from two sources of overhead: data access cost (cost for get/put) and version creation cost (cost for version_inc). As we go to the left-hand side of the graph, where versioning frequency is low, the data access cost dominates. As we go to the right-hand side, where versioning frequency increases, the version creation cost dominates. For reference, we ran the same workload against a flat array but without versioning; this is denoted with the *no versioning* label. Data points at versioning frequency lower than 10^{-5} are missing for the log-structured array because its overhead was too high at these points and it did not complete in a reasonable time.

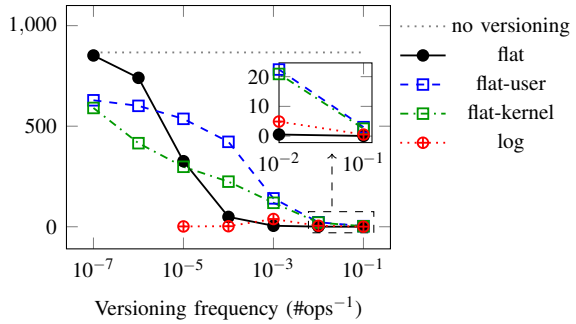


Figure 2. Performance over various versioning frequencies, #procs=32, block size=4096 B, array size=256 MiB/proc, read ratio=50%

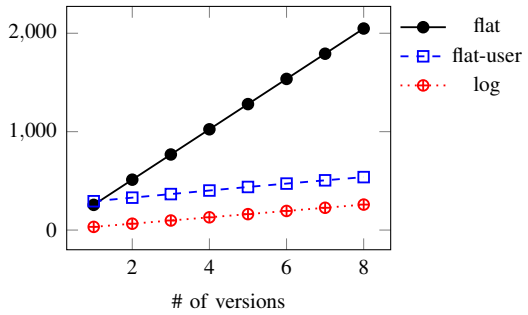


Figure 3. Memory usage comparison; same configuration as in Figure 2, versioning frequency=10^{-5} ops^{-1}

The flat array suffers most from high versioning frequency, because for each version creation it has to copy the entire region of the array. As the versioning frequency decreases, the versioning overhead is amortized, and the performance approaches that of the no-versioning one. The flat with change tracking architecture performs the best in most ranges, since its version creation overhead is lower than that of the flat array because of the smaller amount of data being copied. It is consistently the highest performer for high frequencies ($\geq 10^{-5}$ops^{-1}), beating the second best by over 8 times (10^{-4}ops^{-1}). It does not quite reach the peak performance for lower frequencies, achieving 72% of runs with no versioning, presumably because of the additional communication overheads (see Section III-B). The log-structured array shows moderate performance for the high versioning frequency, but its performance drops heavily when the versioning frequency gets lower. The reason is that although it has an extremely low version creation cost, the log-structured array has a high data access overhead.

D. Memory Usage

We compared the memory consumption of each versioning architecture using our synthetic benchmark as a workload. We ran the benchmark to create 8 versions and

plotted the total memory consumption at the time of each version creation. We counted the following items toward memory consumption: (1) main array buffer (for flat array and flat array with change tracking), (2) array contents that belongs to older versions, and (3) metadata (index) structures to trace data blocks (for flat array with change tracking and log-structured array).

The results are shown in Figure 3. Among all the versioning architectures, the log-structured array consumes the least memory, because it does not have a full array buffer even for the current version. It requires 12.6% of memory compared with that of the flat array. The flat with change tracking architecture (we show the user scheme, but the kernel one performs the same) keeps one full array buffer; thus it consumes the same amount of memory at version 1 as does the flat array. Since it needs to record only modified data blocks, however, the memory consumption increases moderately compared with that of the flat array, at a pace roughly equivalent to that of the log-structured array.

E. Discussion

The flat array with change tracking achieves the best performance in most cases, especially when the versioning frequency is higher than or equal to 10^{-5}ops^{-1}. If the versioning frequency is low, the flat array wins because its high overhead of version creation is amortized. The log-structured array, on the other hand, achieves poorer performance than do the other two, but its memory overhead is extremely low (12.6% compared with that of the flat array). Therefore we conclude that the flat with change tracking architecture is preferable for workloads with moderate or low versioning frequency, whereas the log-structured array can be the best option when versioning frequency is high or when memory saving is the primary concern.

V. Related Work

The log-structured array is based on the idea of the log-structured file system (LFS) [11]. PLFS [12] is an indirection layer that exploits log-structured idea for optimizing checkpoint writes to HPC parallel file systems. These LFSes are designed primarily to improve write performance. However, the log-structured array is designed to capture writes to an array and has incorporated several optimizations for RMA access and multi-versioning, such as fixed-size data block and overwriting an existing block within a single version.

Change tracking uses many of the techniques extensively studied for various checkpoint and recovery schemes, such as in libckpt [13], BLCR [14], SCR [15], and FTI [16], particularly using incremental checkpointing [17].

VI. Summary

In this paper we showed three architectures for array versioning in the Global View Resilience library: flat array, flat array with change tracking, and log-structured array. We compared these architectures using synthetic workloads, in terms of runtime performance and memory overhead. Through a set of benchmark tests we concluded that the flat array with change tracking is preferable in terms of overall runtime performance, whereas the log-structured array would be the best for memory savings.

Future work includes assessing the data recovery cost, data redundancy or vulnerability of each array versioning architecture. Current hardware and software stacks have several limitations that prevents us from fully utilizing the proposed change-tracking schemes. Thus, design, implementation, and evaluation of network hardware devices that support change tracking will be a necessary study.

Acknowledgments

This material was based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, 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] F. Cappello *et al.*, "Toward exascale resilience: 2014 update," *Supercomputing Frontiers and Innovations*, vol. 1, no. 1, 2014.

[2] R. Dreslinski *et al.*, "Near-threshold computing: Reclaiming Moore's law through energy efficient integrated circuits," *Proceedings of the IEEE*, vol. 98, no. 2, pp. 253–266, 2010.

[3] B. Schroeder and G. Gibson, "A large-scale study of failures in high-performance computing systems," *IEEE Transactions on Dependable and Secure Computing*, vol. 7, no. 4, pp. 337–350, Oct 2010.

[4] D. Fiala *et al.*, "Detection and correction of silent data corruption for large-scale high-performance computing," in *SC '12*, 2012.

[5] A. Chien *et al.*, "Exploring versioning for resilience in scientific applications: Global view resilience," in *ICCS*, 2015.

[6] G. Lu, Z. Zheng, and A. A. Chien, "When is multi-version checkpointing needed?" in *FTXS '13*, 2013.

[7] H. Fujita *et al.*, "Log-structured global array for efficient multi-version snapshots," in *IEEE/ACM CCGrid*, 2015.

[8] J. Nieplocha *et al.*, "Advances, applications and performance of the Global Arrays shared memory programming toolkit," *IJHPCA*, vol. 20, no. 2, pp. 203–231, 2006.

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

[10] E. Strohmaier and H. Shan, "Architecture independent performance characterization and benchmarking for scientific applications," in *IEEE MASCOTS*, Oct. 2004, pp. 467–474.

[11] 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.

[12] J. Bent *et al.*, "Plfs: A checkpoint filesystem for parallel applications," in *SC '09*, 2009, pp. 21:1–21:12.

[13] J. S. Plank *et al.*, "Libckpt: Transparent checkpointing under Unix," in *USENIX Technical Conference*, Jan. 1995, pp. 213–223.

[14] P. H. Hargrove and J. C. Duell, "Berkeley Lab checkpoint/restart (BLCR) for Linux clusters," in *Journal of Physics: Conference Series*, vol. 46, no. 1. IOP Publishing, 2006, pp. 494–499.

[15] A. Moody *et al.*, "Design, modeling, and evaluation of a scalable multi-level checkpointing system," in *SC '10*, 2010.

[16] L. Bautista-Gomez *et al.*, "FTI: High performance fault tolerance interface for hybrid systems," in *SC '11*, 2011.

[17] S. Agarwal *et al.*, "Adaptive incremental checkpointing for massively parallel systems," in *ICS*, 2004, pp. 277–286.