Versioning Architectures for Local and Global Memory

Hajime Fujita,*†‡ Kamil Iskra,‡ Pavan Balaji,‡ Andrew A. Chien†‡
*Intel Corporation †University of Chicago ‡Argonne National Laboratory
hajime.fujita@intel.com, 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 that mimics the memory access patterns of radix sort, N-body simulation, and matrix multiplication, comparing the three array architectures in terms of runtime performance, memory requirements, and version restoration costs. 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} ops^{-1} or higher matching the second best architecture or beating it by up to 23 times, whereas the log-structured array is preferable for low memory usage, since it saves up to 98% of memory compared with a flat array.**

***Keywords*-Global View Resilience, multi-versioning, distributed array, change tracking, log-based data structures**

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]. For example, the Blue Waters system had a mean time between failure (MTBF) of 4.2 hours [4]. To make matters worse, future exascale systems are predicted to have MTBF of less than one hour [5]. 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 [6].

To address these issues, we have been developing the Global View Resilience (GVR) library [7]. 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 against latent errors [8], 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 [9] for efficient versioning and provided early exploration of some other versioning architectures [10]. Our current work expands the exploration to cover a hardware-assisted scheme and undo versioning, and provides a significantly expanded evaluation including a comparison of memory change-tracking schemes and experiments with different access localities, read-write ratios, and block sizes, as well as the evaluation of version restoration costs.

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, lowest memory consumption, and lowest version restoration cost, 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.

Specific contributions of this paper are as follows:

- Empirical exploration of trade-offs among three array versioning architectures: flat, flat with change tracking, and log-structured array
- Exploration of local memory change tracking schemes. We found that while the use of the dirty bit in the page tables of Intel® Architecture Processors provides the best performance, reducing overheads on first write access to pages by an order of magnitude compared with using memory protection bits, it is not practical to use with GVR due to not working with DMA and requiring modifications to the OS kernel
- Performance comparison of different versioning architectures across a range of versioning frequencies. We found that the flat array with change tracking achieves the best runtime performance in most cases, for high frequencies ($\geq 10^{-5}$ops^{-1}) matching the second best architecture or beating it by up to 23 times, and for lower

frequencies remaining within 47–97% (depending on the level of locality) compared with a run without versioning
- Measurement of memory overhead of versioning. The log-structured array turned out to be the best solution if memory savings are the primary concern because it saves up to 98% of memory compared with flat arrays, while providing equal or better restoration speed of around 2.15 GiB/s that, unlike the flat with change tracking architecture, is independent of the age of the version to restore from
- Exploration of computational overheads of restoring data from older versions. Flat and log-structured arrays came out on top, besting flat with change tracking arrays by up to 15 times and providing essentially constant access times to every version.

The overall conclusion is that log-structured arrays are preferable for extremely sparse modification patterns (1% fill ratio) in terms of memory consumption and version access cost, whereas the flat with change tracking architecture becomes preferable for more dense modification patterns (10% fill ratio and higher).

The organization of this paper is as follows. In Section II we provide an overview of the GVR library and present a motivating example for seeking efficient multi-version array architectures. In Section III we describe several array versioning schemes. In Section IV we present our evaluation methodology and results from the experiments. In Section V we discuss related work, followed by Section VI where we summarize the work and discuss plans for future work.

II. Multi-Versioning in Global View Resilience

In this section we first give a general overview of the Global View Resilience library, focusing on its multi-version global array. We then present the motivation for implementing efficient multi-version arrays.

A. 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. GVR has two key features: multi-version, multi-stream global arrays and Open Resilience, which allows flexible cross-layer error checking and handling through a unified error-handling interface.

In this paper we focus on the first feature, multi-version 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 [11], 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. More important, the application is responsible for ensuring that an array is in a globally consistent state with respect to the application semantics when taking versions. Thus creating a version also involves an explicit call; applications call *version_inc* function when they want to create a version.

Once a version is created, it is considered read-only. This property makes it easier for the library to transform old versions in various ways, 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. This also means that applications work mainly on the current version rather than on old versions; hence, the runtime performance over the current version is an important metric. Applications can navigate through version history and retrieve data from an arbitrary version using the *get* function. Note that when accessing an old version, the application does not have to retrieve the entire array; it can access only a part. Accessing an old version does not change the contents of the current version, since the result of *get* is returned to a user-supplied separate buffer region.

GVR is implemented as a user-level library built on top of the MPI-3 [12]. 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.

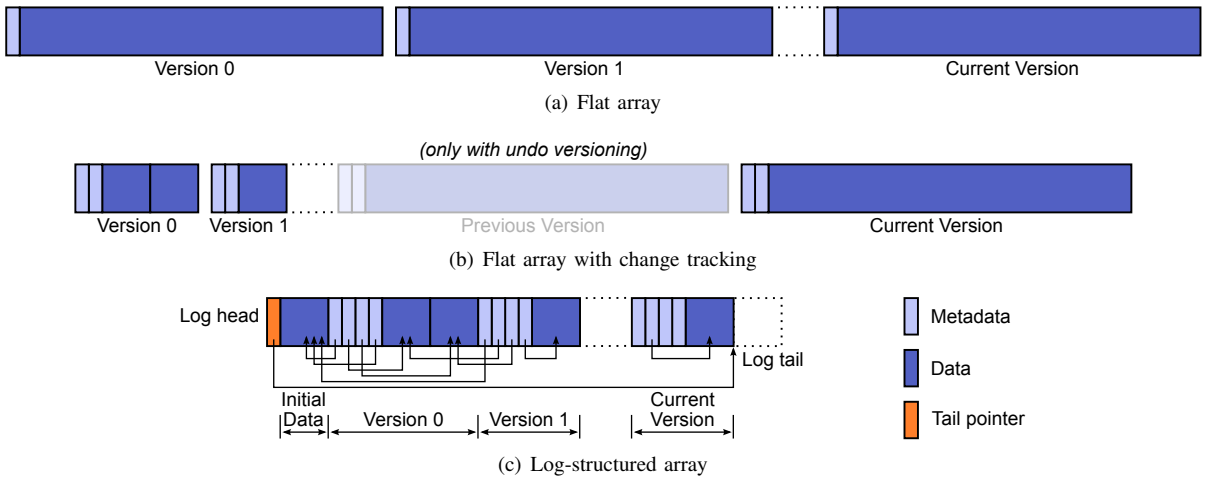
B. How Multi-Versioning Helps

When are multiple versions useful? One significant use case is recovery from latent errors [8]. Traditional checkpoint/restart systems keep only the latest checkpoint, assuming that errors are detected immediately once they occur and checkpoint data is correct. Under the assumption of latent errors, however, this no longer holds. Latent errors might corrupt the data that is going to be dumped to a checkpoint file, before the problem has been discovered. This could result in a corrupted checkpoint; and even if the application discovers the error later, the only way to recover is by restarting the whole computation from the beginning.

Having multiple versions addresses this issue. Even if the latest few versions may be affected by a latent error, a high probability exists that the application can find an older version that is correct.

C. Exploiting Partial Modification for Efficient Versioning

How can such a multi-version array be implemented? The simplest idea is to make a full copy of an array upon version creation. However, our studies found that some application do not modify the whole array region. We studied OpenMC [13], [14], a Monte Carlo simulation for nuclear reactor simulation, and the canneal program in the PARSEC benchmark suite [15], which computes a simulated annealing for circuit design. The modification ratio for OpenMC and



(a) Flat array

(b) Flat array with change tracking

(c) Log-structured array

Figure 1. Data structures of each array versioning architecture

canneal was 24.4% and 2.34% of the array, respectively.[1]

For these types of applications, making a full copy each time makes little sense, since a large fraction of the array does not change. So if we could capture only the modified region, it would save a lot of memory, at the same time enabling faster version creation by reducing the amount of memory to copy. In this paper we study three versioning architectures under this opportunity.

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

As we demonstrate in Section IV, the basic flat array architecture can suffer from high memory requirements and overheads of copying the entire buffer on version creation. We have attempted to reduce such overheads by introducing several change-tracking schemes implemented 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)). Being implemented entirely in user space, this scheme is fairly straightforward; however, it 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 at least the burden factor, since the updates to the bitmap can be hidden inside the put call.

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 OS kernel; and passing such a write-protected buffer to, for example, the `read(2)` system call will result in an error.

A scheme less reliant on signal handlers is needed to overcome this problem, preferably one implemented in hardware for reduced overhead. A suitable option is provided by Intel® Architecture Processors in the form of the page table dirty bit, which is set by the CPU itself on the first write access to a memory page. The bit is, however, already in use internally by the kernel; also, it is not conveniently exposed by Linux* to user space, and no interface is provided to clear it (which we need to do on version creation). Hence, modifications to the Linux* kernel were necessary. We took advantage of the prior work by HP* for Intel® IA-64 architecture, subsequently modified by NCSU for Intel® 64 architecture [16]. We updated the patches to the current Linux* kernel versions and added missing support for huge pages, both regular and transparent. The tracking scheme taking advantage of this capability is called "Hardware."

A concern with the Kernel and Hardware schemes is that

[1]These numbers are much smaller than the ones found in previous work [9], because those numbers were considering fixed-size blocks, whereas numbers here are absolute data sizes.

the page protection bits these mechanisms utilize are guaranteed to work only when the memory is accessed by the CPU. In HPC, the memory may instead be accessed by the DMA engine of the network card, which may not honor the page table bits. Indeed, experiments on our systems (see Section IV-B) have shown that the Hardware bit tracking does not work with DMA (data gets transferred, but dirty bits do not get set). To our surprise, the Kernel scheme does work, but further tests have shown that the latency has increased by an integer factor, and not just for the first access to a page. We found that behind the scenes the InfiniBand* stack refuses to register a write-protected memory buffer we pass at window creation time, falling back to a slower communication mode, presumably utilizing an intermediate buffer. To support the User and Hardware schemes, we implemented separate bit tracking for networking operations. 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. This strategy increases the communication cost, however; ideally, we would like hardware vendors to implement dirty-bit tracking in memory controllers, making it independent of the access method.

Beyond the change-tracking scheme, several parameters influence the LRDS versioning implementation. Block size determines the tracking granularity; for Kernel and Hardware, values below 4,096 (standard page size) make little sense; but for the User scheme, a smaller value can be used. The finer the granularity, the smaller the volume of extraneous data that needs to be copied on version creation (assuming a sparse access pattern), but also the larger the metadata that needs to be maintained. Versioning direction determines whether the data stored in incremental versions can be used for undo or redo operations, which influences the performance of data retrieval (see Section IV-F). In the worst-case scenario, for the redo versioning a retrieval operation may need to traverse all incremental versions back to the oldest version in order to find the required data. When accessing predominantly recent versions, we would instead prefer the undo versioning, where we need to traverse only the incremental versions from the one to retrieve from to the current one. Undo versions, however, increase the memory overhead because, in order to generate them, we need to keep a complete copy of the buffer from the time of the last version creation (Figure 1(b)). This copy is necessary because incremental undo versions need to store the previous contents of modified memory regions, and only after the fact do we learn which regions were modified. Copying in advance could in principle be avoided with the Kernel scheme, where our signal handler is invoked right before the first modification to each page and so could make a copy then; but since code running in the signal handler context must not dynamically allocate memory, the static buffer would need to be allocated in advance anyway.

C. Log-Structured Array

The log-structured array architecture was proposed in previous work [9]. 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). In order to reduce the latency, the log-structured array caches metadata client-side. Creating a new version requires just making a copy of metadata blocks of the latest version. Right after the version creation, all the metadata blocks point to data blocks that were defined in older versions.

More details on the log-structured array can be found in our previous work [9]. Since then we modified the implementation so that it uses MPI_Fetch_and_op for tail pointer manipulation, which brought further performance improvements.

IV. Evaluation

In this section we show empirical comparisons of the three versioning architectures. Evaluations are done based on three criteria: runtime performance in failure-free run, memory consumption, and data retrieval cost from old versions.

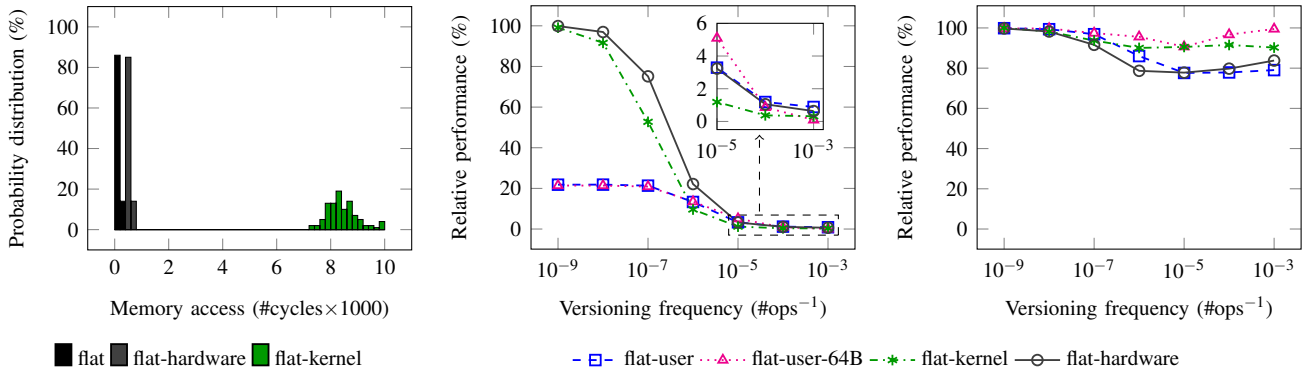
A. Workload

We use a synthetic workload to generate various array access patterns, in terms of versioning frequency, read/write (get/put) ratio, access locality, and so on. Figure 2 shows the kernel of the synthetic application we use.[2] This program

[2]This is similar to the workload used in our previous work [9], with the exception that reads and writes are more finely mixed.

```
while (true) {
  for (i < n_ops_per_version) {
    for (j < n_reads)
      { loc=rndloc(); get(loc, gds); }
    wait(gds); /* Wait for outstanding
        operations to complete */
    n_writes = 10 - n_reads;
    for (j < n_writes)
      { loc=rndloc(); put(loc, gds); }
    version_inc(gds);
  }
}
```

Figure 2. Pseudo-code of the kernel of the synthetic workload



(a) Memory access time on first write to each page, array size=8 MiB

(b) Performance with redo versions relative to no versioning, array size=128 MiB

(c) Performance with undo versions relative to redo ones, array size=128 MiB

Figure 3. Performance of different change-tracking schemes with a local buffer, #procs=1, block size=4096 B (except *flat-user-64B*)

continuously writes to/reads from an array (gds), with an access width of 64 bytes. The target location is randomly determined by the following formula, based on APEX-Map [17]:

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

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 spatial locality of the access sequence. If $k = 1$, the memory access becomes uniformly random. If k is smaller, the memory access is concentrated into a small region. In short, each process makes a large number of array accesses, clustered around the memory region owned by the process. We can tune k to mimic the memory access behavior of several applications. Three values for k—0.0025, 0.025, and 0.25—are used, corresponding to memory access patterns of radix sort, N-body simulation, and matrix multiplication, respectively [17].

B. Setup

We used two platforms, Midway and Breadboard. For most of the experiments we used the Midway cluster installed at the University of Chicago Research Computing Center. Each node has two Intel® Xeon® processors E5-2670 (2.6 GHz, 8-core), 32 GiB RAM, and InfiniBand* FDR-10 as an interconnect. MVAPICH2 2.1rc1 was used as an MPI library. When multiple processes are used, we assigned four processes per node. Because the experiments in Section IV-C1 required a custom kernel, they were instead run on the Breadboard cluster at Argonne National Laboratory, on nodes with two Intel® Xeon® processors E5620 (2.4 GHz, 4-core) and 24 GiB RAM, running Linux* kernel 3.18.4.

The array configurations were as follows:

- *flat*: flat array
- *flat*-{*user,kernel,hardware*}-{*redo,undo*}: flat with change tracking array. Change-tracking scheme (user, kernel, hardware) and versioning direction (redo, undo) follow as needed.
- *log*: log-structured array

To simplify the experiments, we stored all versions in memory.

C. Performance Comparison

1) Fine-Grained Comparison of Memory Change-Tracking Schemes: Because GVR relies on MPI to access the multi-version global array, underlying performance differences between the different change-tracking schemes (see Section III-B) could be masked by communication layer overheads. Hence, we first present the results of experiments when accessing the multi-version array locally (using regular CPU load/store instructions).

For these experiments, we used the RandomAccess kernel from the HPC Challenge suite [18], performing a random walk over a preinitialized memory buffer. In the first experiment (Figure 3(a)) we measured the latency of Kernel and Hardware change-tracking schemes on first write accesses to memory pages. We observe a significant slowdown with the Kernel scheme (close to 8,500 CPU cycles); the Hardware scheme performs much better, at around 550 cycles on average, although that is still far from the performance without tracking (*flat*), measured at around 80 cycles.

Figure 3(b) demonstrates how these overheads affect the performance with versioning enabled, for different frequencies of the latter. The results are relative to the performance without tracking or versioning. For the highest versioning frequencies (10^{-4}–10^{-3}), the overheads of versioning are significant, reducing the performance by two orders of magnitude or more. In this range, the User scheme with 4 KiB block size (*flat-user*) performs best because it has the lowest computational overheads. Kernel and Hardware do worse because they suffer from the overheads of system calls

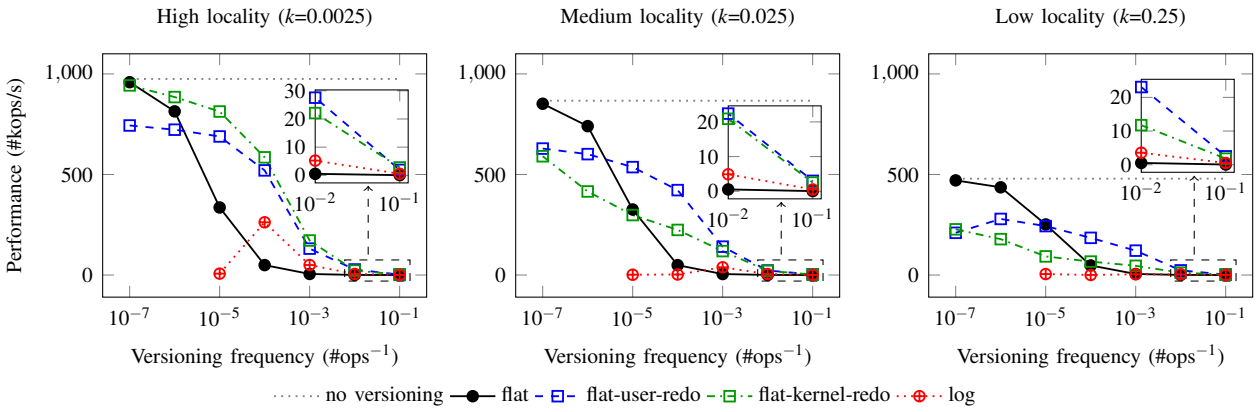


Figure 4. Performance over various versioning frequencies, #procs=32, block size=4096 B, array size=256 MiB/proc, read ratio=50%

and TLB flushes that need to follow the modifications of the page table protection bits. These are per-version overheads, however, so they become less significant with decreasing versioning frequencies. At the frequency of 10^{-6}, Hardware becomes the fastest, followed closely by Kernel; at 10^{-8} each of them achieves over 90% of the performance of a run without change tracking. At such versioning frequencies, the large differences in performance between Kernel and Hardware we observed earlier become less significant because the overheads are experienced only on the first access to each memory page after creating a new version. User schemes, on the other hand, saturate at just over 20%, irrespective of whether we use the block size of 64 bytes (*flat-user-64B*) or 4 KiB. This result is due to the additional overhead of having to mark each buffer modification in the bitmap—a constant overhead independent of versioning frequency.

The results presented above were from the runs that used redo versioning; we ran the same experiments generating undo versions instead and saw largely the same trends. Figure 3(c) compares these two run sets. We see that undo versioning is generally slower, by up to 22%, although that decreases to under 10% for versioning frequencies that could be considered viable (overall overhead under 50%, i.e., frequencies of 10^{-7} and lower). The slowdown is caused by the overhead of additional memory copies at version increment time (see Section III-B).

Overall, these experiments demonstrate that the Kernel and Hardware change-tracking schemes are to be preferred and that the overheads they induce can be overcome but that large amounts of computations between versions are required.

2) Performance of the Entire Software Stack: Then we compared the performance of the synthetic workload (Section IV-A), measured as a total throughput; the results are in Figure 4. 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. For the flat with change tracking configurations we did not conduct the benchmark for the Hardware tracking scheme for several reasons. First, as described in Section III-B, current DMA hardware and software stack do not support change tracking on DMA accesses. Second, as shown in Figure 3(b), even for local accesses the difference between kernel and hardware is minor, and we expect that it becomes much smaller when the overhead of network communications is included. Also we present the results only for redo versioning, because we did not observe significant difference between undo and redo versioning. 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.

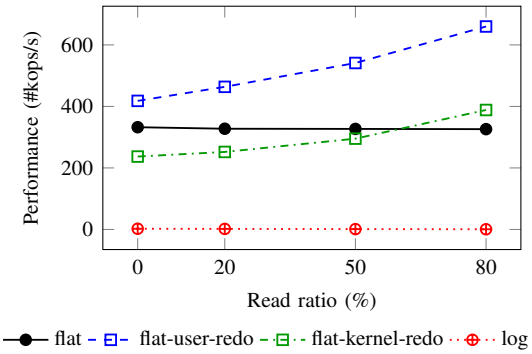


Figure 5. Performance over various read/write ratios, #procs=32, k=0.025 block size=4096 B, array size=256 MiB/proc, versioning frequency=10^{-5} ops^{-1}

Flat array suffers most from high versioning frequency

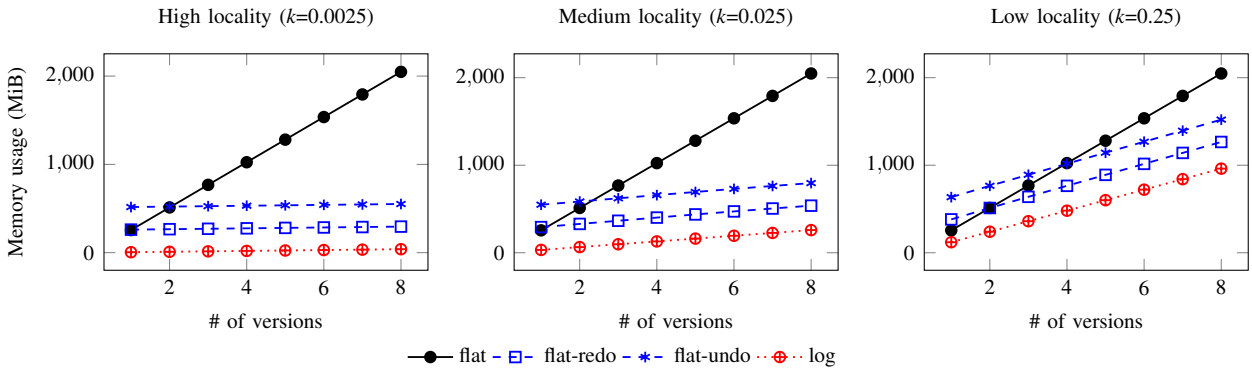


Figure 6. Memory usage comparison; same configuration as in Figure 4, versioning frequency=10^{-5} ops^{-1}

across all access localities (k), 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. Flat with change tracking performs the best in most ranges, since its version creation overhead is lower than flat 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 up to 23 times ($k = 0.25$, 10^{-3}ops^{-1}). It does not quite reach the peak performance for lower frequencies, achieving between 47–97% 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 (get/put) overhead. For all array architectures higher access locality leads to better performance, because it leads to high cache hit rate, and it also reduces overhead for change tracking (for flat with change tracking) as well as data block allocation and metadata management (for log-structured array).

Figure 5 shows how the read/write ratio affects the performance. The read-mostly workload is preferable for flat with change tracking, since writes (puts) require additional dirty-bit accumulation operations. Flat is not affected by the read/write ratio. For the log-structured array we observed the opposite trend, but the performance is almost constant when compared with the other two architectures.

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 and flat with change tracking), (2) array contents that belongs to older versions, (3) metadata (index) structures to trace data blocks (for flat with change tracking and log-structured array), and (4) additional full copy of the main array buffer (for flat with change tracking when using undo versioning).

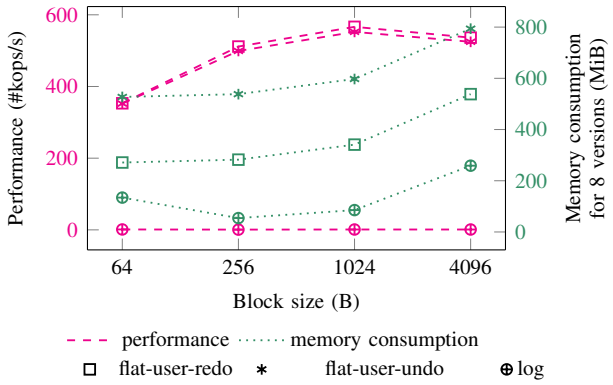


Figure 7. Performance and memory consumption with various block sizes, #procs=32, k=0.025, array size=256 MiB/proc, versioning frequency=10^{-5} ops^{-1}, read ratio=50%

The results are shown in Figure 6. 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 1.9% to 47% of memory compared with flat array, depending on access locality (k). Flat with change tracking architectures keep one (for redo versioning) or two (for undo versioning) full array buffers; thus they consume the same or twice as much memory at version 1 compared with the flat array. Since they need 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. Impact of Block Size

Figure 7 shows how block size affects the performance and memory consumption. Here we compare only flat with change tracking and log-structured arrays, since flat arrays

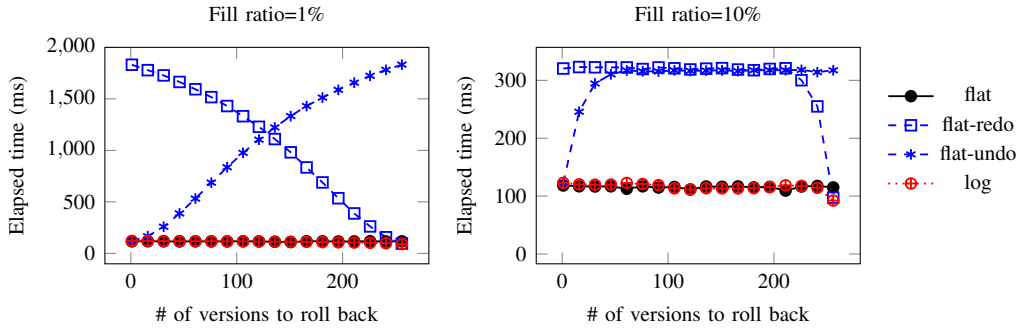


Figure 8. Cost for restoring entire version, #procs=1, block size=4096 B, array size=256 MiB/proc

do not have the block size parameter. The figure generally shows that as block size increases, both performance and memory consumption increase. The reason is that larger block size reduces the overheads of putting data in a data block (e.g., dirty-bit tracking, page fault, etc), while the size overhead also increases. One exception is the memory consumption of the log-structured array at the block size of 64 bytes, where memory consumption is higher than with 256-byte blocks. In this case the relatively small decrease in the data size at that point gets overshadowed by a factor of 4 increase in the number of metadata blocks.

F. Version Retrieval Cost

We measured the version retrieval cost for two scenarios. The first measured the cost of retrieving the whole array at a particular version. This scenario corresponds to a full rollback, for example. The second scenario involved retrieving a small amount of data from an old version, which corresponds to a case where an application performs error checking or a more fine-grained data restoration after a localized memory error.

Before measuring the version retrieval cost, we initialized the array in the following way. We randomly chose a subset of data blocks (say, 10% of the entire array) and put data in them, then created a version and repeated the procedure 256 times, resulting in 256 versions. For versioning schemes other than flat, this initial workload populates a fixed number of data blocks in each version. For this experiment we used a node with 256 GiB of memory so that we could keep all versions in memory. Note that for this experiment, the exact change-tracking scheme for the flat with change tracking array does not matter but the versioning direction (undo/redo) does.

Figure 8 shows the full version retrieval cost. Smaller numbers on the x axis mean more recent versions. The flat and log-structured arrays offer almost constant access time regardless of how old the target version is; around 116 ms, or 2.15 GiB/s. This is not the case, however, with the flat with change tracking arrays. For 1% fill ratio, this architecture can take more than 15 times longer to retrieve than the flat or log-structured arrays do, and still almost 3 times longer for 10% fill ratio (note a different scale on the y axis). Overall, for the flat with change tracking arrays, the average access cost increases as the fill ratio decreases (array becomes more sparse). The reason is that with the flat with change tracking architecture, we conduct a linear search of the version history for a live data block, starting with the version to be retrieved and moving toward the older versions with the redo versioning and toward the newer ones with the undo versioning. Version retrieval cost is proportional to the number of versions traversed. This can be rather high with sparse arrays, but it quickly levels off for the fill ratio of 10%. The reason is that with more dense arrays, there is a high probability of encountering a live block before reaching either end of the version history (for the 10% fill ratio, the vast majority of array blocks require the traversal of fewer than 50 versions). Log-structured arrays, however, do not have this property because they keep the full metadata (index) for each version, so data retrieval cost is independent of the version age. We also conducted the same experiment for the fill ratio of 25% but we did not observe significant difference from 10%, so we do not show it to save space.

Figure 9 shows the cost of small data retrieval. Among 256 versions in total, we chose three versions: the most recent one (1 version old), the oldest one (256 versions old), and the one in the middle (128 versions old). We then picked 10,000 random locations across the entire array, measured the latency of 64-byte data retrieval at each location, and plotted a cumulative distribution function of data retrieval time. The overall trend is similar to what we saw with the full version retrieval. The flat and log-structured arrays offer nearly constant access times, whereas the flat with change tracking arrays show both a higher average and variation with the 1% fill ratio. We again see clearly that the undo versioning works better when retrieving from a recent version and that the redo versioning works better when retrieving from an old version. For the fill ratio of 10% and 25%, differences between the flat with tracking and other architectures become less significant, thus we did not show these results due to space limitation.

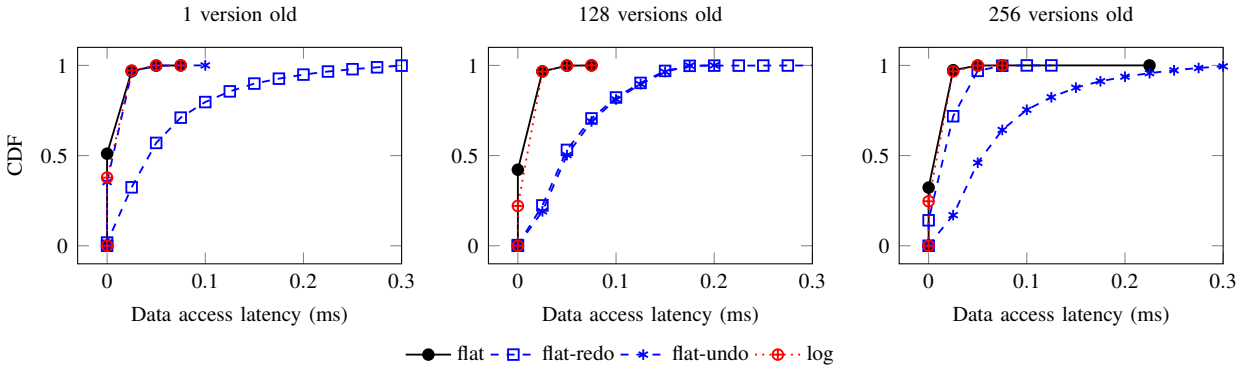


Figure 9. Cost for restoring random locations, data access size=64 B, #procs=1, block size=4096 B, array size=256 MiB/proc, 1% fill ratio

G. Discussion

Our experiments enable us to make several observations.

1) Performance: The flat array with change tracking achieves the best performance in most cases, especially when 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 the other two, but its memory overhead is extremely low (1.9% to 47% compared with that of flat). We also observe that the flat array with change tracking suffers from poor version retrieval performance with extremely sparse array modification patterns (1% fill ratio), whereas the flat and log-structured architectures do not. Therefore we conclude that the flat with change tracking architectures is preferable for workloads with moderate or low versioning frequency (also implies moderate array fill ratio), whereas the log-structured array can be the best option when versioning frequency is high or when memory saving is the primary concern.

2) Controlling Redundancy for Resilience: Our experiments show that the flat with change tracking and log-structured arrays save significant amounts of memory, because they do not preserve the data that was not updated. However, this also means that these architectures reduce redundancy, which may have a negative impact on resilience. We argue that we can add redundancy back as needed. For the simplest example, we can create and keep a copy of each data and metadata block; the total memory consumption would still be less than or comparable to that of the flat array. Moreover, we can easily optimize the degree of redundancy, for example, by creating a redundant copy only once in several versions or by applying error-correcting code for space efficiency. However, detailed design and analysis are beyond the scope of this paper and are left as future work.

V. Related Work

The log-structured array is based on the idea of the log-structured file system (LFS) [19]. PLFS [20] is an indirection layer that exploits log-structured idea for optimizing checkpoint writes to HPC parallel file systems. These LFSs are primarily designed 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.

Recent studies of distributed key-value data stores have also been exploring log-structured data, such as RAMCloud [21], [22], SILT [23], and Tango [24]. Significant differences between these distributed key-value stores and the multi-version array in GVR are twofold: (1) the GVR array is addressed by multi-dimensional indices, and (2) it has a property that once a data block is assigned in a version, subsequent writes to the same block do not require additional allocations until the next version.

Change tracking uses many of the techniques extensively studied for various checkpoint and recovery schemes [25], such as in libckpt [26], BLCR [27], SCR [28], and FTI [29], with incremental checkpointing [30], [31], [32] being of particular importance. Our Hardware change-tracking scheme directly benefits from the work of Vasavada et al. [16]. Undo versioning benefits from earlier work on reverse computation, with optimistic parallel discrete-event simulation being one of the early applications [33]. More recently, reverse [34] and replay [35] debugging has renewed the interest in the field. Doudalis and Prvulovic proposed Euripus [36], a unified hardware acceleration for both bidirectional debugging and checkpoint/restart applications—something we could definitely benefit from to reduce the overheads of our change-tracking schemes. Unfortunately, contemporary mainstream hardware fails to implement such acceleration techniques.

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, memory overhead, and version retrieval cost. Through a set of benchmark tests we concluded that the flat with change tracking array is preferable in terms of overall runtime performance, whereas the log-structured array would be the best if the modification pattern is quite sparse and memory saving or version restoration cost is the primary concern.

Future work includes assessing the data redundancy or vulnerability of each array versioning architecture and adding redundancy back to the array, as discussed in Section IV-G2. Current hardware and software stacks have several limitations that prevents us from fully utilizing the proposed change-tracking schemes, especially the Hardware-based change tracking. 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 RCC.

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] C. Di Martino *et al.*, "Lessons learned from the analysis of system failures at petascale: The case of Blue Waters," in *IEEE/IFIP DSN*, 2014, pp. 610–621.

[5] M. Snir *et al.*, "Addressing failures in exascale computing," *IJHPCA*, 2013.

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

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

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

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

[10] ——, "Empirical comparison of three versioning architectures," in *IEEE Cluster*, 2015, (to appear).

[11] 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.

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

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

[14] N. Dun *et al.*, "Data decomposition in Monte Carlo neutron transport simulations using global view arrays," *IJHPCA*, 2015, (to appear).

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

[16] M. Vasavada *et al.*, "Comparing different approaches for Incremental Checkpointing: The showdown," in *Proceedings of the Linux Symposium*, Ottawa, ON, Canada, Jun. 2011, pp. 69–79.

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

[18] P. Luszczek *et al.*, "The HPC Challenge (HPCC) benchmark suite," in *SC'06*, Tampa, FL, 2006.

[19] 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.

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

[21] D. Ongaro *et al.*, "Fast crash recovery in RAMCloud," in *SOSP '11*, 2011, pp. 29–41.

[22] S. M. Rumble, A. Kejriwal, and J. Ousterhout, "Log-structured memory for DRAM-based storage," in *FAST 14*. USENIX, 2014, pp. 1–16.

[23] H. Lim *et al.*, "SILT: a memory-efficient, high-performance key-value store," in *SOSP '11*, 2011, pp. 1–13.

[24] M. Balakrishnan *et al.*, "Tango: Distributed data structures over a shared log," in *SOSP '13*, 2013, pp. 325–340. [Online]. Available: http://doi.acm.org/10.1145/2517349.2522732

[25] E. N. Elnozahy *et al.*, "A survey of rollback-recovery protocols in message-passing systems," *ACM Computing Surveys*, vol. 34, no. 3, pp. 375–408, Sep. 2002.

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

[27] 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.

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

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

[30] J. S. Plank, J. Xu, and R. H. B. Netzer, "Compressed differences: An algorithm for fast incremental checkpointing," University of Tennessee, Tech. Rep. CS-95-302, Aug. 1995.

[31] R. Gioiosa *et al.*, "Transparent, incremental checkpointing at kernel level: A foundation for fault tolerance for parallel computers," in *SC '05*, 2005.

[32] S. Agarwal *et al.*, "Adaptive incremental checkpointing for massively parallel systems," in *ACM International Conference on Supercomputing (ICS '04)*, 2004, pp. 277–286.

[33] D. R. Jefferson, "Virtual time," *ACM Transactions on Programming Languages and Systems*, vol. 7, no. 3, pp. 404–425, Jul. 1985.

[34] B. Boothe, "Efficient algorithms for bidirectional debugging," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '00)*, 2000, pp. 299–310.

[35] S. Narayanasamy, G. Pokam, and B. Calder, "BugNet: Continuously recording program execution for deterministic replay debugging," in *International Symposium on Computer Architecture (ISCA '05)*, 2005, pp. 284–295.

[36] I. Doudalis and M. Prvulovic, "Euripus: A flexible unified hardware memory checkpointing accelerator for bidirectional-debugging and reliability," in *International Symposium on Computer Architecture (ISCA '12)*, 2012, pp. 261–272.