ERDC/ITL MP-22-1

US Army Corps
of Engineers®
Engineer Research and
Development Center

In Situ Visualization with Temporal Caching

David E. DeMarle and Andrew C. Bauer

January 2022

Information Technology Laboratory

The U.S. Army Engineer Research and Development Center (ERDC) solves the nation's toughest engineering and environmental challenges. ERDC develops innovative solutions in civil and military engineering, geospatial sciences, water resources, and environmental sciences for the Army, the Department of Defense, civilian agencies, and our nation's public good. Find out more at www.erdc.usace.army.mil.

To search for other technical reports published by ERDC, visit the ERDC online library at https://erdclibrary.on.worldcat.org/discovery.

ERDC/ITL MP-22-1
January 2022

In Situ Visualization with Temporal Caching

Andrew C. Bauer

Information Technology Laboratory
US Army Engineer Research and Development Center
3909 Halls Ferry Road
Vicksburg, MS 39180

David E. DeMarle

Intel Corporation
2200 Mission College Blvd
Santa Clara, CA, 95054

Final report

Prepared for US Army Corps of Engineers
Washington, DC 20314

Under Program Element 633465, Project Number AL2, Task SAL301

Preface

This study was conducted for the US Army Corps of Engineers (USACE) under the High Performance Computing Modernization Program (HPCMP), Program Element Number 633465, Project Number AL2, Task SAL302.

The work was performed by the US Army Engineer Research and Development Center, Information Technology Laboratory (ERDC-ITL). At the time of publication of this paper, the deputy director of ERDC-ITL was Dr. Jackie Pettway and the director was Dr. David Horner.

This article was originally published in *Computing in Science & Engineering* on 29 March 2021 (Revised 15 June 2021).

COL Teresa A. Schlosser was the commander of ERDC and the director was Dr. David W. Pittman.

In Situ Visualization With Temporal Caching

Abstract: In situ *visualization is a technique in which plots and other visual analyses are performed in tandem with numerical simulation processes in order to better utilize HPC machine resources. Especially with unattended exploratory engineering simulation analyses, events may occur during the run, which justify supplemental processing. Sometimes though, when the events do occur, the phenomena of interest includes the physics that precipitated the events and this may be the key insight into understanding the phenomena that is being simulated.* In situ *temporal caching is the temporary storing of produced data in memory for possible later analysis including time varying visualization. The later analysis and visualization still occurs during the simulation run but not until after the significant events have been detected. In this article, we demonstrate how temporal caching can be used with in-line* in situ *visualization to reduce simulation run-time while still capturing essential simulation results.*

With the ever increasing compute power available to today's simulation scientists, the amount of data potentially produced is overwhelming. Data management is listed as one of the top ten exascale challenges.[1] One method for reducing the data deluge in scientific workflows is through the use of *in situ* analysis and visualization, which is the process of computing-derived quantities of interest from a simulation during the computation. The earliest known example of *in situ* processing was done by Zajac[2] in 1964 in generating a video directly from a simulation run. Since then a variety of work has been done in the field.[3,4]

One of the concerns with *in situ* processing is that by saving reduced data products, e.g., images, data extracts, statistical quantities, etc., information may be lost and a full understanding of the simulation results may not be obtainable as when saving full data dumps. This can be alleviated with automatic feature detection methods combined with increased resolution data extraction and tracking following the event detection. A wide variety of work has been done in domain specific feature detection and extraction methods, and a proper literature review is beyond the scope of this discussion. Instead, a notable recent domain-agnostic work was done by Larsen *et al.*[5] One the issue with performing feature detection *in situ* is that all current work has used forward stepping algorithms. Thus, anything "interesting" that occurred before the feature was detected is no longer available and cannot be recovered for *posthoc* use.

Our focus in this article is on the concept of in-line *in situ* temporal caching, where simulation outputs are temporarily copied to fast storage for possible later use, in combination with *ex postfacto* triggers. These triggers are conditional tests that induce the production of analysis results that utilize the cached data. Together they extend the ability of *in situ* feature detection to reduce the likelihood of information loss. The technique is particularly useful in compute bound cases where the simulation alone does not stress system memory capacity. When memory use is near the limit, software reduction strategies like downsampling and compression can help, as can intermediate storage hardware technologies such as on-node NVDIMMs and off-node burst buffers. These software and hardware technologies support temporal caching with differing impacts on the run-time cost. High-performance computing systems with dedicated large memory nodes, for example, the Intel Optane Memory enabled nodes in the Texas Advanced Computing

Center's Frontera supercomputer with terabytes of space per node, are ideal platforms for exploring the technique. We demonstrate the effectiveness of in-line *in situ* temporal caching to reduce simulation run-time while also outputting the results of interest by instrumenting temporal caching inside of the ParaView Catalyst *in situ* library and manually implementing triggers inside of the *in situ* run-time control routines. This is showcased with example problems from two separate simulation codes.

DESIGNS

Two main structural patterns exist for designing *in situ* frameworks. In *in transit* systems, data are moved to separate processing units for concurrent processing while the simulation is running. For *in-line* systems, the simulation and analysis routines are more tightly coupled, using the same processing elements and usually living within the same process and memory space. A popular example of *in transit* processing is the ADIOS[6] framework. ParaView Catalyst[7] is a popular choice more aligned to *in-line* processing, where the simulation drives visualization algorithms in a direct, tightly coupled data sharing arrangement. In both patterns nontrivial examples, that is beyond simply generating animations, of temporal *in situ* processing exist but temporal processing is more established with *in transit,* so we begin there.

In Transit Solutions With Temporal Caching

In transit systems have made good use of burst buffers and HPC file IO for some time now. For example, Hamilton *et al.*[8] demonstrated ways in which the burst buffer helps with both data reduction and compression for reduced file IO costs.

With *in transit* systems, since the data are conceptually and often physically copied between systems, it is a straightforward next step to build in temporal buffering. Doing so helps to avoid synchronization between producing and consuming processes. For example, in Zanúz *et al.*,[9] a molecular dynamics mini-app was connected through Apache Flink to an HBase database on the disk space of a dedicated set of nodes. While the mini-app ran, Flink embedded spatial histogramming and nearest neighbor computations were run on each timestep, and both the input and derived data were fed into the data base. The authors determined the simulation rate at which message buffers began to slow the postprocessing without slowing the simulation over the run. Once in the database of course, cross temporal *posthoc* visualization of the data across timesteps could be handled naturally, but this was not a focus.

Another example of loosely coupled *in situ* visualization with temporal buffering is PaDaWAn.[10] Here, the simulation's IO calls are intercepted and substituted for network calls. The network calls are designed to funnel the data into a staging area. The staging area can be indexed by both the producing rank and data timestep. Client processes can subscribe to the staging area and retrieve data for analysis concurrently with the executing simulation code. The examples given for the system were time stepping animations and downsampling to reduce disk output. Temporal information was specifically included with *in situ* processing for later *posthoc* analysis through reduction techniques such as by Marsaglia *et al.*,[11] but there is little published literature on temporal processing entirely within the *in situ* run.

Temporal Caching In-Line

ParaView is an open-source framework for large-scale scientific visualization. ParaView is based on The Visualization Toolkit (VTK), which provides the underlying data processing and rendering capabilities. ParaView, when packaged as a library that is connected to and driven by a simulation code, is called Catalyst. Traditional Catalyst setups consist of a simulation code extended with an adaptor that periodically feeds data into one or more *in situ* processing pipelines that produce extracts (rendered images or geometry files) from the newly generated data. The adaptor's function is to convert in-memory simulation data structures into in-memory vtkDataObject data structures, often making use of efficient zero-copy reference passing.

A variation of Catalyst's design that supports temporal *in situ* processing is illustrated in Figure 1. This option was added in Catalyst version 5.9. When enabled, the adaptor's vtkDataObjects are managed by VTK's temporally aware vtkTemporalDataSetFilter class. This class implements a ring buffer that retains data from multiple timesteps, providing them on demand to the rest of the visualization pipeline. Note that in Catalyst, the contents of the vtkDataObjects are typically transient; passed by reference, used for visualization, and then released after each iteration. For temporal processing, we must instead make a copy of the data provided by the simulation on every iteration, as we presume that the simulation data itself will either be deleted or updated with new results in the next iteration. This means that the minimum memory cost to use our system is approximately two times that of the base simulation.

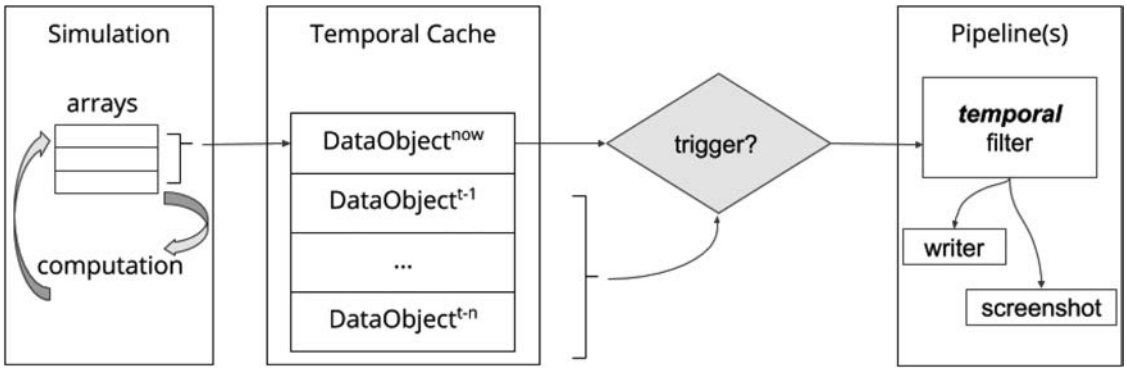

FIGURE 1. Temporal processing of *in situ* produced data. New results are copied into a fixed size in memory result buffer for possible reuse in response to preprogrammed conditions, should they arise during the simulation run.

Besides this change to buffer the simulation's data, changes were also made to give ParaView more control over where data resides. ParaView now has a rudimentary out of core memory system built at the VTK level, which allows all VTK-based applications to work directly with data arrays that are resident on the file system. This optional component depends on the memkind library.[12] Memkind is a heap manager library for various types of memory that gives absolute control to a developer in regards to allocating memory from an application. The library allocates and manages memory on different physical medium (high bandwidth memory, DRAM, persistent memory, or memory mapped disk space) per the application's request.

From a VTK application developer's perspective, the key entry points to taking advantage of the extended memory system are to initialize memkind and to create objects intended to be out of core by calling vtkObject::ExtendedNew() instead of vtkObject::New(). The source of the selected data depends upon the initialization parameters to memkind.

With the caching system in place, an extended triggering mechanism is needed to detect interesting events and react to them. ParaView Catalyst preferentially uses Python as the control language. Python provides plenty of flexibility to detect and react to arbitrary events and reduces the coding effort required to do so. Figure 2 demonstrates a simple Catalyst Python trigger, extracted from TemporalCacheExample in ParaView's source code. In the full example, a toy simulation moves particles around a box and resamples them onto a regular grid. The trigger is a detected collision event by two or more particles.

The advantage of adding a caching system to inline *in situ* is that the analysis code can be more selective about which time steps to process. In essence, it becomes more similar to *posthoc* processing where a user may see an interesting feature and step back in time to see what led up to that feature. For the *in situ* case, we rely on preprogrammed triggers to detect these interesting features rather than on user interaction. The capability is potentially efficient because only when features are detected do we actually do significant processing.

RESULTS AND USE CASES

Some of the expected use cases that drove our combined temporal cache with *ex postfacto* triggering work include the following.

- Simulation debugging where we wish to reversibly debug data conditions that infrequently arise and eventually lead to invalid results.
- Simulation runs of physical phenomena that we want to rewind and expose the causes of.

```
def ProcessTriggers(self, datadescription):
  # access the just produced data
  data = self.Pipeline.data
  # check for a condition of interest
  oRange = data.GetArrayInformation("occupancy").GetRange()
  if oRange[1] > 1: # more than one in some cell, ie a collision
    # when the event happens react, in this example
    # by writing the entire cache to long term storage
    writer = writers.XMLPImageDataWriter(Input=data)
    for time in data.TimestepValues:
      writer.SetFileName("tstep_"+time+".pvti")
      writer.UpdatePipeline(time)
```

FIGURE 2. Pseudocode for an example *ex postfacto* trigger. In practice more bookkeeping is required for technicalities like parallel aggregation of the trigger condition and not overwriting timesteps. See the full example in the ParaView source code for details.

- Design tradeoff or case study runs where multiple simulations with slight input variations lead to exceptional events that occur in only a subset of the simulation runs. A real-world example is of exhaust gas reingestion for rotorcraft, where hot exhaust gas recirculates back to the intake of an engine and causes deleterious effects under certain design and flight conditions.

In general terms, temporal caching can be thought of as extending the usual case of *in situ* processing with a support of one timestep to a larger support with many timesteps.

- With a support of one, pure *in situ* visualization is relegated to creating animations and saving extracts for later *posthoc* processing.
- There are numerous cases where a support of two, that is computations that involve both the previous and current timestep, are useful. Some examples are: interpolating between timesteps, displaying temporal derivatives to emphasize locations of rapid change, and recovering constant time spacing from adaptively time stepped inputs.
- Moving further there are operations that are enhanced with a fixed buffer size greater than two, for example, visualizing streaklines and pathlines, reducing high-frequency noise with running averages, minimizing disk storage by omitting expected results, and the aforementioned *ex postfacto* use cases.
- In the limit, there is the case when the hardware has excess memory resources compared to the simulation's needs and the buffer size can be said to be practically unlimited. This simplifies many of the abovementioned use cases and opens new ones such as periodicity determination.

To illustrate the effectiveness of triggers and temporal caching at reducing the execution time during an *in situ* simulation run, we introduce the concept of *in situ* computational load. Computational load helps to quantify the variation in computation time that the *in situ* processing requires over the entire simulation run. We define *in situ* computational load as the time to compute all *in situ* products at one time step divided by the total time until the next iteration that *in situ* processing is started. This is shown in Figure 3. The principle idea is that when *in situ* computational load is low, or at zero for much of the run then we know that the detection triggers are doing a proper job. Because we are caching data and executing the trigger algorithm, we do not expect the *in situ* computational load measure to be zero but do expect it to be efficient relative to other parts of the simulation.

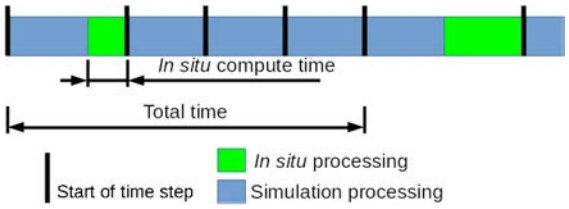

FIGURE 3. Graphic representation of *in situ* computational load showing *in situ* compute time compared to total time.

Performance Evaluation

We ran variations of ParaView's TemporalCacheExample to evaluate the performance of its new cache on different hardware technologies. In the runs, we sized the problem so that each timestep produces 1 GB of data so as to exceed the test machine's total DRAM capacity near the end of the run. We ran traditional configurations of: the simulation without visualization, the simulation with core adaptor code executing but no additional processing, and the simulation running and calling the adaptor to write data files to disk. We compared these with temporal caching runs that keep either 50 or 450 time steps worth of data in the cache, where the cache is resident on DRAM, on Optane RAM, or on memory-mapped files. Furthermore, we compared two boot modes for the operating system, a two level memory (2LM) mode where the operating system configures DRAM to be a Level 4 cache under Optane, and an App Direct (AD) mode in which the application is responsible for managing Optane and DRAM resident data manually. Our test platform has 384 GB of DDR4 DRAM, 3 TB of Intel Optane persistent memory, and 15 TB of 3-D NAND SSD (P4510).

Table 1 describes the results. Here, we recorded average compute plus cache times over the run. Early in development, our initial results were somewhat disappointing with caching times that were on par with the cost of writing data to disk (which neglects the cost of reading back data, i.e., traditional *posthoc* visualization). Profiling indicated that almost all of the overhead came from the deep copy operation that fills the cache. This overhead is unavoidable for temporal caching. We found that by threading the deep copy operation, we could take better advantage of Optane's bandwidth characteristics. The thread-optimized

TABLE 1. Measurements of a temporal caching system's performance.

Run configuration	Seconds per time step		
	m=500 *n*=0	*m*=500 *n*=50	*m*=500 *n*=450
Simulation only	0.46	NA	
With Adaptor	0.46	NA	
Write to NAND	1.17	NA	
Threaded Cache to Optane 2LM	NA	0.55	0.60
Threaded Cache to DRAM	NA	0.52	out of space
Threaded Cache to Optane AD	NA	0.59	0.66
Threaded Cache to NAND	NA	0.95	1.21

Shown is the average compute plus caching only time over an "*m*" timestep run With an "*n*" timestep wide cache on a toy simulation.

results of 0.55 and 0.60 s per time step adds only 0.09 and 0.14 s of overhead, respectively, to the core code cost, which is in practice negligible to the compute time of many simulations that are run on HPC systems today.

In comparing the AD and 2LM modes, we expected to find that explicitly managed AD mode was more performant but the results showed otherwise. The slight benefits observed with the 2LM mode may be an artifact of routing data through the faster DRAM layer.

Second-Order Hydrodynamic Automatic Mesh Refinement Code (SHAMRC)

The SHAMRC code is a second-order accurate, multiphysics code used to study nuclear and conventional blast effect phenomenology. This code was a good candidate for Catalyst integration, as many SHAMRC calculations typically run with billions of computational cells, where generating animations that have a high temporal fidelity can be prohibitive. SHAMRC has been used to study a wide range of effects on blast phenomenology such as dust sweep up in nuclear environments and the response of highly energetic materials to the blast environment. Frequently, many phenomena of interest to the modeler happen after a well-defined event has occurred. For example, in SHAMRC's energetics material model, a shock reflection might cause increased material mixing leading to a rise in aerobic burning of detonation products, or it could reheat dispersed metallic particles in a measurable fashion. Additionally, variability in factors such as shock speed due to shock energy or superposition with other shock waves can vary the time when events of interest occur. These factors provide good motivation to include *ex postfacto in situ* capabilities in SHAMRC.

SHAMRC has been tested with the Catalyst's *ex postfacto* visualization capability on a simple planar shock wave interacting with a raised block. Our run was done on the US Department of Defense's Cray XC40, Onyx, with 22 MPI processes. The temporal caching was set to store five time steps worth of datasets and the frequency of this was set to every 20th cycle. The Python script has a trigger that checks for an increased pressure at the corner of the raised block and produces graphics at that cycle, the cached cycles before as well as cycles after the detection that satisfy the trigger criteria. For our simulation, the first detection in the *in situ* script occurred during the 700th time step. During the *in situ* processing at this cycle, images were output for the 600th, 620th, 640th, 680th, and 700th cycle, with only the last one not being from cached data. The images for this along with the trigger location are shown in Figure 4.

OpenFOAM

OpenFOAM is a widely used open-source simulation library for a variety of continuum mechanics solvers that is managed and supported by ESI-OpenCFD Ltd. OpenFOAM is a popular choice for computational fluid dynamics problems. OpenFOAM has included a Catalyst adaptor framework since version v1806.

We used OpenFOAM's rotating propeller tutorial in three sets of runs to test out temporal caching on Frontera's Intel Optane Memory enabled nvdimm nodes. This tutorial was chosen because it is representative of the type of actual work that OpenFOAM is used for, is easily recognizable, and can be scaled up to be more computationally challenging. We ran the simulation on Frontera in various configurations from one rank on one node to 512 ranks spread across eight nodes. For the runs we cached both the internal (vector field) and boundary (propeller and boundaries).

First, a comparison of two visualization algorithms helps to illustrate how visualizing results across time steps rather than as a simple series can illuminate aspects of the data in different ways. The first are streamlines, in which velocity fields (the simulated fluid) are rendered by advecting traces throughout the field at each time step. The next are pathlines, which are paths that particles take as they move within the field as it changes over time. It should be noted that *in situ* solutions to pathline generation are not novel but

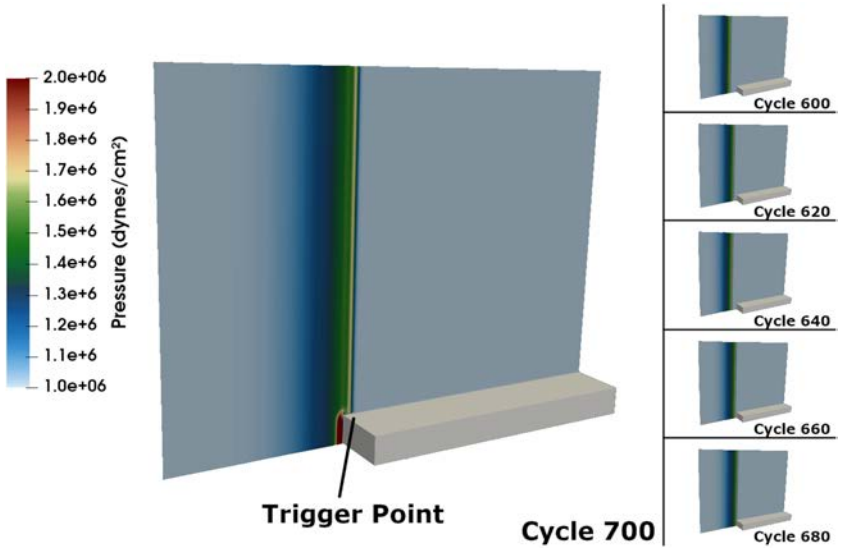

FIGURE 4. Computed pressure level at a location in space during Cycle 700 triggered visualization of the simulation at the current iteration along with previous iterations leading up to the event.

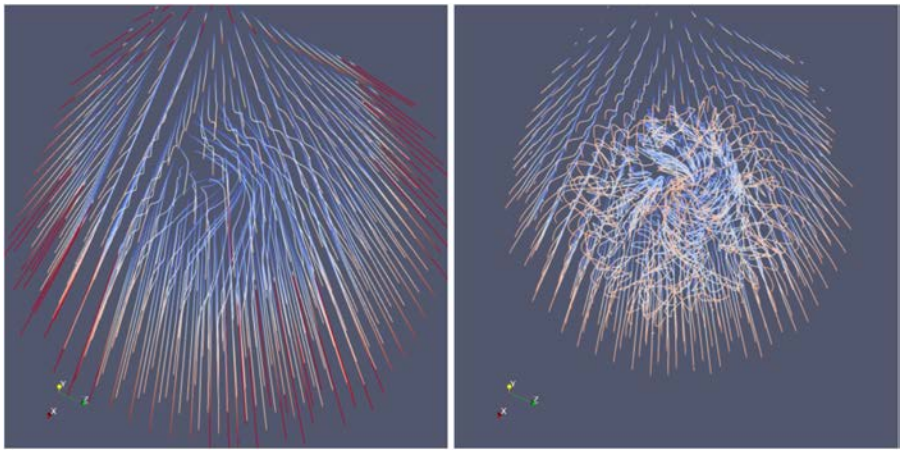

FIGURE 5. Instantaneous streamlines (left) and time-integrated pathlines (right) emphasize different aspects of the vector field.

that the general framework we have provided facilitates the implementation of this and many other visualization operations. The images in Figure 5 were created midway through the run. The fluid flow is generally toward the camera and slightly downward. The pathline image conveys the turbulent region more strongly and conveys the fact that flow has progressed only partially through the volume more clearly.

A second set of OpenFOAM propeller runs helps to characterize expected performance of a full featured simulation code on production hardware. For these runs, we vary the *in situ* configuration and benchmark the cycle time throughout each run. In all runs, we execute for 1,000 cycles evenly spaced over 0.1 s of simulated data time. The first configuration is the baseline and is without any visualization or IO. A second configuration renders the propeller, streamlines, and an isosurface of the turbulence on every tenth cycle without caching. A third configuration exercises temporal caching throughout the run and has the trigger condition satisfied at cycle 500. When the trigger condition is satisfied, the *in situ* processing is generated for the 10 preceding timesteps and then subsequently at every 10th timestep. The rendering for the third case is identical to the second case.

The trigger setup here is somewhat arbitrary but meant to be a median representation of what users

TABLE 2. Measured run times in seconds for the openfoam propeller simulation.

Run configuration	Full run time	*In situ* compute time
Simulation only	109	NA
Simulation with *in situ* output every 10th step	148	39
Simulation with temporal caching and *in situ* output	134	25

The simulation ran for 1,000 timesteps. The temporal caching setup cached every time step until the trigger was activated at timestep 500.

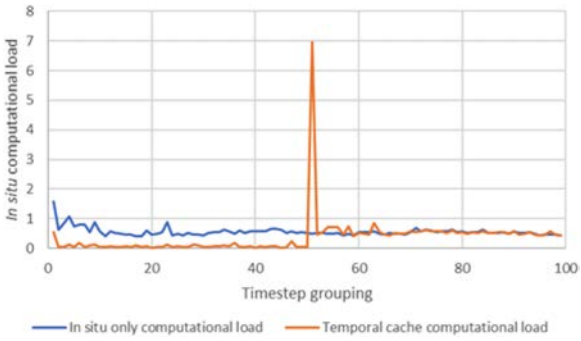

FIGURE 6. *In situ* computational load for runs with and without temporal caching. The timestep grouping is the set of timesteps that include only a single *in situ* output. For our sample problems the timestep grouping size is 10, which is the timestep frequency of the *in situ* output.

may experience from multiple sample problems. It would be trivial to set up a sample problem that would show temporal caching in the most favorable conditions but this would be of little instructional use.

Table 2 describes the total time for each run and the *in situ* processing time for each run (for temporal caching this includes copying data to cache and computing the trigger). While this showcases the savings that can be obtained by using *in situ* temporal caching compared to outputting *in situ* results throughout the entire simulation, it does not make it clear where the savings come from. Figure 6 shows how the *in situ* computational load varies during the simulation run for both *in situ* examples. Both have start up costs but then the temporal caching technique reduces costs significantly by evaluating the trigger and caching data instead of computing the full *in situ* output. There is a spike in the cost for the temporal cache case when the trigger evaluation criteria is satisfied and the visualization pipeline is executed on the cached timesteps. This spike is minimal though when compared to the overall savings.

In general, we find that caching on cycles without visualization is competitive with raw simulation time and that analysis of some number of cycles after the fact is competitive with analyzing the same number of cycles without caching. In practice, the expected run-time will vary most with the frequency of detected events, the amount of disk space conserved will vary inversely, and the most important run-time concern is the increased need for memory space.

A final set of runs with the propeller case served to exercise more of the capacity of Frontera's large memory nodes. Each nodes has 2.1 TB of Optane memory. To approach the capacity of four nodes, we increased the mesh refinement and decreased the simulation timestep significantly and ran until we reached a data time of 0.25 s instead of 0.1 as above. For this run, the trigger condition was simply set to fire at two particular data times, (0.11 and 0.22 s). At those trigger points, we play back the entire contents of the cache to generate new views of the data. At the end of the run, the cache contains all 7,667 timesteps. In both replays, we change the isosurface extracted from the turbulence field, the camera viewpoint, and various color controls. These are all options that a user might use in a *posthoc* analysis while investigating time varying data. Images produced during the run are shown in Figure 7. Note also that we turned ON Intel OSPRay path traced rendering to stress the compute capability of the nodes and create more visually interesting images.

CONCLUSION

In situ visualization is an area of active research within the scientific visualization community. In this field, non-trivial time varying analyses that cut across the time dimension seem to be underutilized. We have presented several cases in which a simple sliding buffer of cached results could make this type of temporal analysis more commonplace, even in in-line *in situ* runs.

Temporal buffers, combined with *ex postfacto* triggers and reactions can also serve as the foundation for unattended exploratory visualization. In this case, the visualization solution is programmed to adapt to handle unexpected situations more intelligently. The benefit of temporal buffering in this situation is that the trigger handling routines have access to potentially important computed data that would otherwise have been lost.

Looking forward, we expect that for multirun use, for example, in ensemble runs, uncertainty quantification analysis, etc., *in situ* analysis and visualization will become a key tool in reducing time to engineering solutions. When running hundreds, thousands, or more runs,

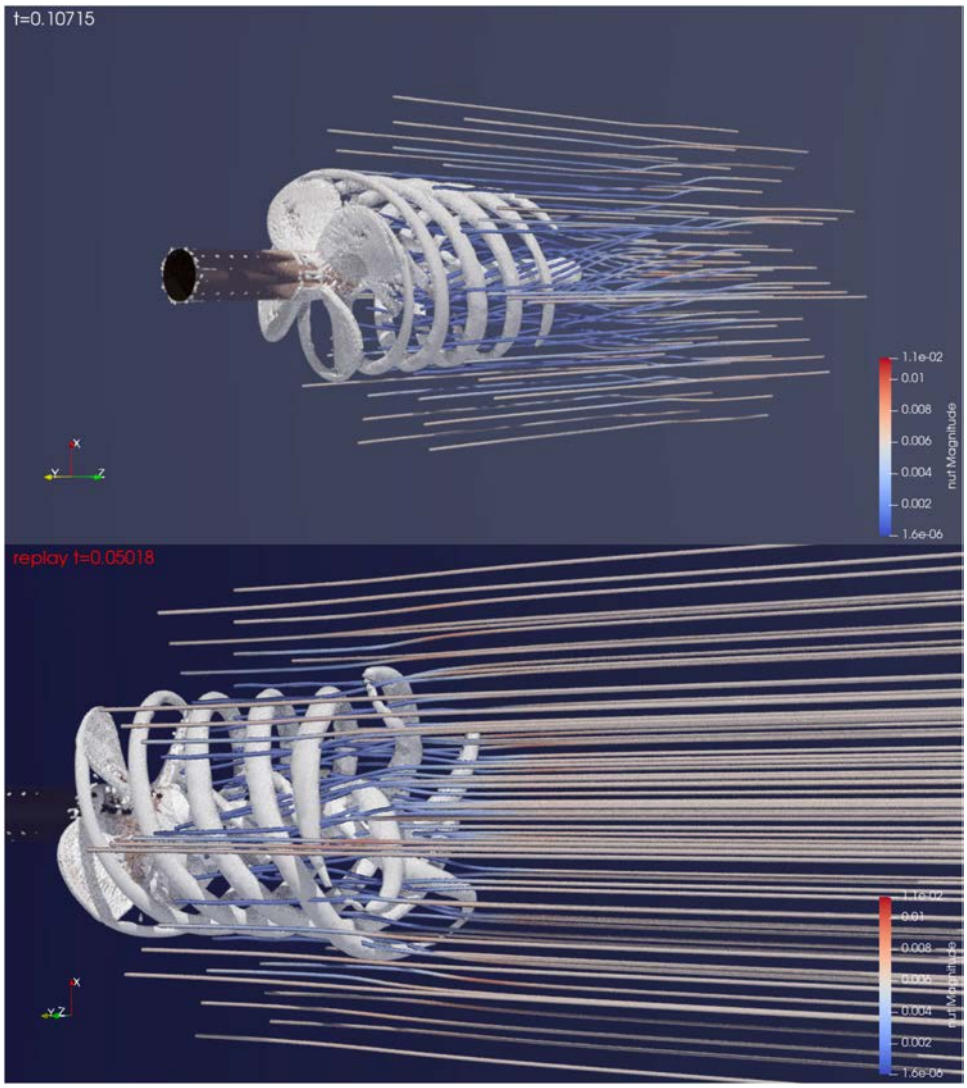

FIGURE 7. Snapshots from a temporal cached *in situ* run. The top was visualized with new data obtained while the cache was being filled, the bottom was visualized from data in the cache.

the user will not be able to manually analyze every output in-depth. Instead, the analysis will need to be automated and the analysis will need to be prepared before the multirun starts. In this situation, the aspect of reducing disk space use by bypassing the export of unimportant time steps will be especially beneficial.

ACKNOWLEDGMENTS

This work was made possible by the Department of Defense High Performance Computing Modernization Program's allocation of compute time on the Engineer Research and Development Center's Department of Defense Supercomputing Resource Center's Cray XC40 Onyx, and with compute time on the Texas Advanced Computing Center's Frontera supercomputer. The authors would like to thank Joseph. Hennessey of SAIC and David. Ortley of ARA, Inc. with help in running SHAMRC, Michal. Biesek for advice with memkind, and Paul. Navrátil of TACC for assistance on Frontera.

REFERENCES

1. R. Lucas *et al.*, "DOE Advanced Scientific Computing Advisory Subcommittee (ASCAC) Report: Top Ten Exascale Research Challenges," ASCAC Subcommittee for the Top Ten Exascale Research Challenges, Tech. Rep., Jul. 2013, [Online]. Available: https://science.osti.gov/ascr/ascac/Reports, doi: 10.2172/1222713.

2. E. E. Zajac, "Computer-made perspective movies as a scientific and communication tool," *ACM Commun.*, vol. 7, no. 3, pp. 169–170, Mar. 1964, doi: 10.1145/363958.363993.
3. R. Heiland and P. M. Baker, "A survey of co-processing systems," CEWES, Rep. no. 99–02, 1999. [Online]. Available: https://rheiland.github.io/coproc/CoprocSurvey.pdf
4. A. C. Bauer *et al.*, "In situ methods, infrastructures, and applications on high performance computing platforms," *Comput. Graph. Forum*, vol. 35, no. 3, pp. 577–597, Jun. 2016, doi: 10.1111/cgf.12930.
5. M. Larsen *et al.*, "A flexible system for in situ triggers," in *Proc. Workshop In Situ Infrastructures Enabling Extreme-Scale Anal. Vis.*, 2018, p. 1–6, doi: 10.1145/3281464.3281468.
6. H. Abbasi, J. Lofstead, F. Zheng, K. Schwan, M. Wolf, and S. Klasky, "Extending I/O through high performance data services," in *Proc. IEEE Int. Conf. Cluster Comput. Workshops*, Sep. 2009, pp. 1–10, doi: 10.1109/CLUSTR.2009.5289167.
7. U. Ayachit *et al.*, "Paraview catalyst: Enabling in situ data analysis and visualization," in *Proc. 1st Workshop In Situ Infrastructures Enabling Extreme-Scale Anal. Vis.*, Nov. 2015, pp. 25–29, doi: 10.1145/2828612.2828624.
8. S. Hamilton *et al.*, "Extreme event analysis in next generation simulation architectures," in *Proc. High Perform. Comput.: 32nd Int. Conf., ISC High Perform.*, Jun. 18-22, 2017, pp. 277–293, doi: 10.1007/978-3-319-58667-0_15.
9. H. C. Zanúz, B. Raffin, O. A. Mures, and E. J. Padrón, "In-transit molecular dynamics analysis with apache flink," in *Proc. Workshop In Situ Infrastructures Enabling Extreme-Scale Anal. Vis.*, 2018, pp. 25–32, doi: 10.1145/3281464.3281469.
10. J. Capul, S. Morais, and J.-B. Lekien, "Padawan: A python infrastructure for loosely coupled in situ workflows," in *Proc. Workshop In Situ Infrastructures Enabling Extreme-Scale Anal. Vis.*, 2018, pp. 7–12, doi: 10.1145/3281464.3281470.
11. N. Marsaglia, S. Li, and H. Childs, "Enabling explorative visualization with full temporal resolution via in situ calculation of temporal intervals," in *High Performance*, R. Yokota, W. Weiland, J. Shalf, and S. Alam, Eds. Cham, Switzerland: Springer, 2018, pp. 273–293, doi: 10.1007/978-3-030-02465-9_19.
12. C. Cantalupo, V. Venkatesan, J. R. Hammond, K. Czurylo, and S. Hammond, "Memkind: An extensible heap memory manager for heterogeneous memory platforms and mixed memory policies," Sandia Nat. Lab. (SNL-NM), Albuquerque, NM (United States), 2015.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED *(From - To)*
January 2022	Final	

4. TITLE AND SUBTITLE

In Situ Visualization with Temporal Caching

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER
633465

6. AUTHOR(S)
David E. DeMarle and Andrew C. Bauer

5d. PROJECT NUMBER
AL2

5e. TASK NUMBER
SAL301

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
U.S. Army Engineer Research and Development Center
Information Technology Laboratory
3909 Halls Ferry Road
Vicksburg, MS 39180

Intel Corporation
2200 Mission College Blvd
Santa Clara, CA 95054

8. PERFORMING ORGANIZATION REPORT NUMBER
ERDC/ITL MP-22-1

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
U.S. Army Corps of Engineers
Washington, DC 20314

10. SPONSOR/MONITOR'S ACRONYM(S)
USACE

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES
This article was originally published in *Computing in Science & Engineering* on 29 March 2021 (Revised 15 June 2021).
Funding by USACE ERDC under Army Direct funds.

14. ABSTRACT
In situ visualization is a technique in which plots and other visual analyses are performed in tandem with numerical simulation processes in order to better utilize HPC machine resources. Especially with unattended exploratory engineering simulation analyses, events may occur during the run, which justify supplemental processing. Sometimes though, when the events do occur, the phenomena of interest includes the physics that precipitated the events and this may be the key insight into understanding the phenomena that is being simulated. In situ temporal caching is the temporary storing of produced data in memory for possible later analysis including time varying visualization. The later analysis and visualization still occurs during the simulation run but not until after the significant events have been detected. In this article, we demonstrate how temporal caching can be used with in-line in situ visualization to reduce simulation run-time while still capturing essential simulation results.

15. SUBJECT TERMS
Computational modeling; data visualization; data models; catalysis; feature extraction; numerical analysis; analytical models; cache storage

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			19b. TELEPHONE NUMBER *(include area code)*
Unclassified	Unclassified	Unclassified	UU	14	