# Towards High Performance Resilience using Performance Portable Abstractions

Nicolas Morales, Keita Teranishi, Bogdan Nicolae, Christian Trott, Franck Cappello

# Towards High Performance Resilience using Performance Portable Abstractions

Nicolas Morales[1], Keita Teranishi[1], Bogdan Nicolae[2], Christian Trott[1], and Franck Cappello[2]

[1] Sandia National Laboratories, USA
{nmmoral,knteran,crtrott}@sandia.gov
[2] Argonne National Laboratory, IL, USA
{bnicolae,cappello}@anl.gov

**Abstract.** In the drive towards Exascale, the extreme heterogeneity of supercomputers at all levels places a major development burden on HPC applications. To this end, performance portable abstractions such as those advocated by *Kokkos*, *RAJA* and *HPX* are becoming increasingly popular. At the same time, the unprecedented scalability requirements of such heterogeneous components means higher failure rates, motivating the need for resilience in systems and applications. Unfortunately, state-of-art resilience techniques based on checkpoint/restart are lagging behind performance portability efforts: users still need to capture consistent states manually, which introduces the need for fine-tuning and customization. In this paper we aim to close this gap by introducing a set of abstractions that make it easier for the application developers to reason about resilience. To this end, we extend the existing abstractions proposed by performance portability efforts towards resilience. By marking critical data structures that need to be checkpointed, one can enable an optimized runtime to automate checkpoint-restart using high performance and scalable asynchronously techniques. We illustrate the feasibility of our proposal using a prototype that combines the *Kokkos* runtime (HPC performance portability), with the *VELOC* runtime (large-scale low overhead checkpoint-restart). Our experimental results show negligible performance overhead compared compared with a manually tuned implementation of checkpoint-restart while requiring minimal changes in the application code.

**Keywords:** Performance Portability · Resilience · Fault Tolerance · Checkpointing · Programming Models

## 1 Introduction

Supercomputing facilities have seen a rapid evolution towards heterogeneous architectures, both from a computational (many-core CPUs, GPUs, other accelerators) and I/O perspective (deep memory hierarchies, node-local persistent storage, external parallel file systems, key-value stores, etc.). This places a major

burden on HPC applications, because they need to interface with a large variety of vendor APIs and/or customize their codes accordingly.

In an effort to address this problem, *performance portability* [8] has been proposed as a potential solution. The key idea of performance portability is the abstraction of hardware heterogeneity behind a unified programming model that eliminates the need for customization on the application side, while shifting the awareness of the intrinsic aspects of different heterogeneous accelerators to the runtimes, which can transparently provide optimized implementations of the high-level abstractions for each type of accelerator. Efforts such as *Kokkos* [8], *RAJA* [13], *DPC++* [20] are just some examples that illustrate this concept.

Unfortunately, the increasing number of heterogeneous components driven by the quest to achieve Exascale also leads to an increasing failure rate, which means resilience is another important challenge that needs to be addressed by the applications. Since most of the applications running on HPC machines are tightly coupled, failures are hard to isolate and quickly propagate from one process to another. In many cases, error detection can be more limited on heterogeneous nodes (such as on a GPU) [5]. Therefore, the main resilience strategy used by these applications is global checkpoint/restart where all processes agree periodically on a globally consistent state that is persisted and used to restart from in case of failures.

While there are many optimized checkpointing frameworks available that combine a variety of techniques (e.g. asynchronous multi-level resilience strategies as illustrated by *VELOC* [19]) to reduce the checkpointing overhead as much as possible, such frameworks typically require the application developers to manually assemble critical data structures and serialize them into node-local files, which are then persisted to resilient storage. On heterogeneous nodes, this is a non-trivial task. First, some data structures may live on GPUs or other accelerators and are not accessible in the host space. Secondly, there may be multiple references to the same data structure that only needs to be checkpointed once. Finally, a data structure may need to be part of the critical state in one location of the code but not in another (e.g., some data structures may be discarded at the end of a main loop). Therefore, there is a need to simplify how application developers reason about resilience in order to address the aforementioned issues without sacrificing the ability to leverage optimized checkpointing frameworks.

To this end, we propose a set of resilience abstractions for heterogeneous architectures that hide the complexity of interacting with checkpoint-restart frameworks, similar to how performance portability approaches hide the complexity of interacting with heterogeneous hardware. By synergizing with such efforts, we leverage the fact that users already define their data structures using constructs such as *memory views*. These can be extended and used together with another construct, the *scoped resilient execution contexts*, allowing the runtime to both capture the critical data structures and checkpoint them at the right moment automatically. This unique combination provides an intuitive and efficient way for performance-portable applications to employ resilience, reducing devel-

oper time and cost, while enabling a straightforward integration with optimized checkpointing frameworks.

We summarize our contributions as follows: first, we introduce a novel resilience model specifically designed to take advantage of performance portability to automate both how to capture and when to checkpoint critical data structures. Second, we show how to implement such a model in practice based on the *Kokkos* [8] and *VELOC* projects [19]. Finally, we run extensive experiments to demonstrate the benefits of our proposal, both using synthetic benchmarks and real-life HPC applications.

## 2 Related Work

The move towards Exascale and heterogeneous computing platforms has increased the complexity of HPC hardware and software systems. Various studies have explored the effects of extreme-scale on application resilience. Di Martino et al. [5,4] analyze the resiliency of 5 million HPC application runs on the Blue Waters supercomputer, considering both CPU compute nodes and CPU nodes with GPUs and recording the hardware and software failure rates over the better part of a year. Hukerikar and Engelmann [14] lay out a set of resilience design patterns for large scales and a literature survey of the field.

*Multi-level checkpoint/restart* is a popular approach to leverage multiple storage levels in the context of HPC checkpointing. Works representative of this approach include SCR [18] and FTI [2], which introduce support for local storage, partner replication, erasure coding (XOR and Reed-Solomon) and finally external storage (parallel file systems). Recent efforts such as *VELOC* can take advantage of heterogeneous storage for each level and introduce advanced asynchronous techniques that leverage synergies between the levels [19] and predictions of application behavior to mitigate interference [23].

A growing field of user-level software resilience techniques has emerged around the idea of enabling MPI processes to repair the communicators (instead of terminating the whole MPI deployment), which can be leveraged to implement forward recovery techniques. Examples in this direction are the ULFM extension to MPI [15,22] and user-level programming models such as Fenix [24,12,9,10], as well as the Relax transactional framework proposed by De Kruijf et al. [3].

Other areas of software fault tolerance focused on extreme scale and heterogeneous systems include resilience for task-parallel programming models [17,21], GPU snapshotting approaches [1], and the DataSpaces staging service [6] along with resilience strategies for multi-application staging [7]. Additionally, there has been recent focus on local checkpointing and rollback functionality to prevent a global rollback over the entire system [11,16,22].

All the approaches mentioned above require significant development effort either to capture and serialize the global state as checkpoints (rollback recovery) periodically, or to reconstruct consistent states after failures in other ways (forward recovery). To the best of our knowledge, we are the first to explore the idea of leveraging performance portability abstractions as building blocks

for a resilience model that bridges the gap between application development productivity and other resilience strategies.

## 3 Background

The goal of performance portability is to provide HPC applications with a unified programming model that allows code to be written once and reused for a diverse array of architectures with similar performance. Several efforts have been proposed in this space, including *Kokkos* [8], *RAJA* [13] and *DPC++* [20]. These frameworks provide abstractions for parallel/concurrent computation and memory/data representation for heterogeneous computing systems to hide platform specific programming features and complex interactions between the host CPUs and the accelerators. Under these frameworks, individual data objects (e.g. arrays) encapsulate the metadata information such as device location and memory layout. Adding a unique identification to the metadata enables to track the state of individual data objects, facilitating an automation of application-based checkpointing and recovery. For the purpose of this work, we focus on *Kokkos*. However, it is important to remember that our approach does not depend on any particular performance portability framework and can be adapted accordingly.

To achieve this goal, *Kokkos* introduces three abstractions: (1) *execution spaces*, which define how computational kernels can be executed at fine granularity in a data-parallel fashion; (2) *memory views*, which define multi-dimensional arrays that computational kernels operate on; (3) *polymorphic data layout*, which enables memory views to be reinterpreted dynamically.

In terms of execution spaces, simple *Kokkos* patterns look similar to those exposed by OpenMP, the main difference is the focus on a C++-oriented model. Of interest to us is how these patterns interact with the data structures, which are captured as memory views. Specifically, the memory views can be annotated with properties and hints that refer to their layout, location (host or device), and access pattern. This enables the application to fix accessibility and performance relationships. For example, a memory view can be interpreted in different ways when being processed on a CPU or GPU: extra copies can be avoided if it is known to be read-only (*const*) or a computation in the GPU execution space could access data directly from the host memory (e.g. using unified virtual memory) with degraded performance.

## 4 Performance Portable Resilience Abstractions

Starting from the performance portable abstractions discussed above, this section introduces the core design principles of our proposal.

***Scoped resilient execution contexts:*** As mentioned above, the *Kokkos* execution spaces (e.g., parallel for) require the user to encapsulate the computational

kernels into lambda functors. We extend this notion to provide a higher-level resilient execution context. These contexts are encapsulated into lambda functions and can encompass one or more *Kokkos* operations that exist in any execution space. Using this approach, any external memory views used in the body of the lambda function are automatically captured when the context is closed, forming a set of critical data structures that can be checkpointed and used to restart. In turn, the resilient execution context will handle all other aspects automatically: when to checkpoint, how to serialize the critical memory views, and so on. An example of how this works is illustrated in Listing 1. It is important to note that this approach grants high levels of flexibility: users can simply compose multiple scoped resilient execution contexts, each with its own lambda captures and therefore an implicit minimal set of critical memory views. Therefore, users are freed from manually keeping track of what data structures are critical in all alternative paths of their code. This would normally be a cumbersome process that is both prone to errors (for example, if the user forget to checkpoint a critical data structure) and sub-optimal performance (checkpointing a data structure that is not critical).

***Optimized tracking of critical memory views:*** Our goal is to avoid introducing a new construct that forces the developers to differentiate between "regular" and critical memory views, which is why users are allowed to capture regular memory views in resilient execution contexts. When it is time to checkpoint, a naive approach would be to serialize all captured memory views into a checkpoint file and continue the execution. However, such an approach is inefficient for several reasons. First, memory views may be reinterpreted or copied in multiple execution contexts, but only one instance is enough to reconstruct the state on restart. In this case, we allow memory views to be marked as *transient* (checkpoint not required) or *aliased* by a unique identifier. Aliasing views allow the runtime to track two views with different names as the same. This is primarily useful for adapting codes where a view is replaced by another that is functionally the same but has a different allocation. An example of this would be a double-buffering algorithm, where two views are named differently but are swapped so only one is mutable at a time. While aliasing can be used to ensure correctness of some code, it isn't necessary in most cases. Marking a view as transient does not affect correctness, but can provide some performance benefits by reducing checkpointing size.

All views are reference counted intrinsically; a memory view is checkpointed only once even if it is captured multiple times. The combination of aliasing and reference counting create a de-duplication opportunity that can significantly reduce the checkpoint sizes independently of the checkpoint backend. Furthermore, another optimization opportunity lies in the fact that memory views may be read-only, which means either a different execution context was responsible for generating it or can simply be regenerated on restart. Our approach can detect what views are read-only based on their "*const*-ness". This can be leveraged to take advantage of checkpointing backends that implement incremental checkpointing techniques.

```
const int dim0 = 5, dim1 = 5;
auto view = Kokkos::View< double ** >( "test_view", dim0, dim1 );

for ( int iter = 0; iter < max_iter; ++iter ) {
  KokkosResilience::checkpoint(plugin, "test_checkpoint", iter, [=]() {
    Kokkos::parallel_for( dim0, KOKKOS_LAMBDA( int i ) {
      for ( int j = 0; j < dim1; ++j )
        view( i, j ) = 3.0;
    } );
  } );
}
```

Listing 1: Example usage of scoped resilient execution context.

***Dynamic pluggable checkpointing backends:*** Once the minimal set of critical memory views was determined, our approach can transparently interface with any checkpointing backend that was specifically optimized for a particular scenario and/or machine. The translation from our unified model to the various APIs of the checkpointing backends are implemented as independent plugins that can be flexibly assigned by the users to each resilient execution context. In fact, it is perfectly possible to mix different checkpointing backends in the same application or even dynamically switch between them. Although outside of the scope of this paper, it is important to note that this capability can be leveraged as a building block for more advanced checkpointing approaches, such as dynamic decisions based on the size and/or content (such as applying compression algorithms if the memory views are large and/or sparse).

## 5 Implementation

The resilience layer functions on two levels. The first level operates at compile time and encompasses the definition of the resilient execution contexts and the detection of views along with their type properties such as *const*-ness. The second level is the runtime component that tracks view usage (as defined in the first layer), aliases, and the interface with the checkpointing backend.

***Defining the resilient execution context:*** In order to specify the actual critical regions of code that must be made resilient through checkpoint/restart, one must define the beginning and the end. In general, code in the resilient region should have no side effects. If code inside the region were to modify global or static variables, these would not be included in the checkpoint and could affect correctness of the program during a restart. Under these assumptions, scoped resilient execution contexts can be defined as lambda captures, as mentioned in Section 4.

Listing 1 shows a simple usage scenario, where `plugin` is the resilience plugin, `test_checkpoint` is the label and `iter` is the iteration number to be checkpointed. Finally, the scoped resilient context is specified as a lambda with input and output views part of the lambda capture. Note that the user does not need to explicitly

```
Kokkos::View< double * > ping( /*...*/ ), pong( /*...*/ );
for ( int i = 0; i < max_ts; ++i ) {
  Kokkos::View< const double * > read;
  Kokkos::View< double * > write;
  if ( i % 2 )
    read = pong; write = ping;
  else
    read = ping; write = pong;
  KokkosResilience::checkpoint( ctx, "iterate", i, [=]() {
    Kokkos::parallel_for( /*...*/, KOKKOS_LAMBDA( int j ) {
      write( j ) = do_calculation( read );
    } );
  } );
}
```

Listing 2: Example of a ping-pong buffer for which a minimal set of critical memory views is automatically detected by examining *const*-ness of all captured views.

remember to capture a view; it is automatically captured by value. This incurs a minimal overhead and does not create extra copies, because *Kokkos* views are constructed as references. In the case that existing code makes extensive use of global views, it is possible to explicitly capture the global variable (or make reference counted copies of the global that are implicitly used within the lambda) to bring it under the scope of the resilience tracking algorithms.

***Tracking of critical memory views:*** One of the primary features of our approach, as outlined in the previous section, is the optimization of *const Kokkos* memory views. Normally a view has a datatype associated with it (such as `double`). If the datatype is instead *const*, such as `const double`, the view becomes immutable. *Kokkos* already provides conversions from mutable to *const* views that work as expected; similarly these views share the same reference count and control block. We take advantage of this fact in our implementation; when a view is only used in a checkpoint region for reading, marking it as *const* will prevent the runtime from needing to checkpoint the view. This can be determined at compile time, when the lambda capture list is built. This is straightforward with C++ function overloading. List-building is delegated to two functions, one taking a templated *const* view type, and another taking a non-*const* view type. In this way, *const* views can be filtered from the checkpoint list. Of course, this relies on some input by the user in marking *const* views; however this is generally well-accepted practice both in C++ programming in general and *Kokkos* code. We apply the same principle for transient views, while aliased views are de-duplicated at runtime.

To illustrate this point, consider the following example in Listing 2: a buffer that is written to changes every iteration. However, only one buffer should be actually checkpointed, since the other one is redundant. Since the read buffer is not written to and is *const*, it can be ignored. On a restart the write buffer would be loaded and immediately swapped into the new read buffer. This automatic optimization would, in this example, lead to a 50% reduction in checkpoint size as opposed to checkpointing every view.

***Checkpointing backend integration:*** Our research prototype implements a plugin for *VELOC* [19], a low overhead checkpointing runtime specifically designed to deliver high performance and scalability for HPC machines thanks to a combination of multi-level resilience strategies that are applied asynchronously. *VELOC* is a particularly well suited candidate for illustrating our proposal because it implements a memory-oriented API that separates the registration of critical memory regions from the actual checkpointing operations, thereby simplifying the integration.

Specifically, we automatically register and unregister the *Kokkos* memory views with *VELOC* as contiguous memory regions defined by pointer and size whenever this is possible, which gives *VELOC* direct memory access and therefore reduces the checkpointing overhead. However, in the case of views with non-contiguous layout or views in inaccessible memory (e.g., GPU memory) that cannot be directly registered/unregistered with *VELOC*, we have implemented a proxy layer that collects all non-contiguous and/or inaccessible memory regions into a host buffer, which in turn is then registered with *VELOC*. At the start of any subsequent checkpoint, the runtime synchronizes with the backend and completes any pending asynchronous operations.

In addition to *VELOC*, we have implemented a simple synchronous approach that writes the checkpoints directly to a parallel file system. Based on this initial set of plugins, we plan to interface with several other backends.

## 6 Results

### 6.1 Experimental setup

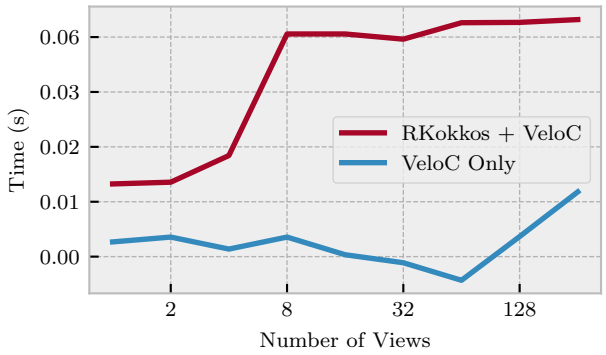


**Fig. 1.** Stress-test results for a weak scalability experiment that involves an increasing number of checkpointed memory views. The comparison refers to the bookkeeping overhead related to tracking the memory views using our approach vs. manual registration using *VELOC*. Checkpointing is deactivated, but tracking is still active. This timing data includes compute time.

Our experiments were carried out on an experimental testbed at Sandia National Laboratories featuring 1488 compute nodes, each equipped with 2.1 GHz Intel Broadwell CPUs (36 cores, 72 hardware threads). The memory of each node is 128 GB of RAM. Omni-Path interconnect is used between the nodes, which is exposed through OpenMPI v.4.0. All compute nodes have access to a Lustre parallel file system to persist their data.

In terms of software ecosystem, we forked Kokkos v.3.0 and added our scoped resilient execution abstraction on top of it. The *VELOC* checkpoint plugin was written for *VELOC* v.1.4, which features a client library and an encapsulated resilience engine that can be either linked with the application as a synchronous checkpointing library, or as a separate service in an active backend that enables asynchronous support. Our experimental setup leverages the latter to improve the checkpointing efficiency. Furthermore, *VELOC* was configured to use a `tmpfs` in-memory filesystem based on shared memory (`/dev/shm`) in order to minimize the overhead of capturing local checkpoints, which are then flushed asynchronously to Lustre.

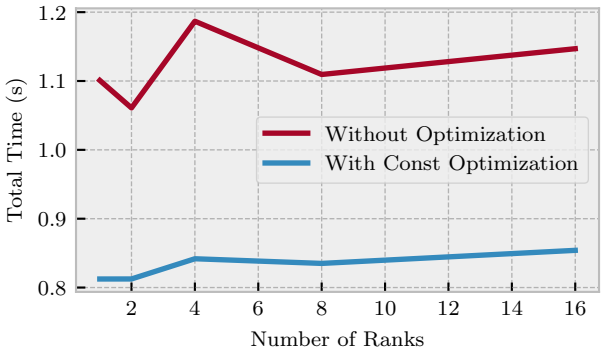


**Fig. 2.** Weak scaling of the *ping-pong* microbenchmark (Section 5) with and without the *const* memory view tracking optimization. The checkpoint size is 50% smaller when the const-tracking is activated, which significantly improves the checkpointing performance.

### 6.2 Methodology

To evaluate our proposal, we devise a series of experiments that mix both synthetic benchmarks and real-life HPC applications.

Specifically, we designed two micro-benchmarks to evaluate particular aspects of our implementation in order to understand potential limitations and bottlenecks in extreme cases. These will be discussed in the rest of this section along with the results. In addition, we use two scientific mini-apps:

***MiniMD:*** is a parallel molecular dynamics application written using Kokkos abstractions for the Mantevo [3] project. We forked the code and implemented resilience using the abstractions of our proposal, as discussed in Section 5.

***HeatDis:*** is a heat distribution solver (HeatDis) that is used as an example by *VELOC*[4]. First, we parallelized HeatDis with *Kokkos*, but kept the original resilience implementation that calls *VELOC* directly. Next, we re-implemented the resilience in the Kokkos-enabled HeatDis using our own abstractions.

We conducted experiments at scale using multiple nodes, each of which leverages the available cores using OpenMP-based *Kokkos*, which was configured to use 70 threads. Although *VELOC* is designed to prevent interference with the applications during asynchronous checkpointing, we decided to eliminate this noise from our experiments by allocating the remaining core (2 hardware threads) on each compute node exclusively to the *VELOC* active backend.

### 6.3 Results: Microbenchmarks

First, we ran a series of experiments to understand the overheads that are introduced by our approach due to tracking of the memory views. To this end, we compare our approach with a baseline that relies on manual registration of the critical data structures using *VELOC*. To emphasize this overhead as much as possible, we do not perform an actual checkpoint (they are converted to no-ops), which means the runtime directly measures the bookkeeping overhead of tracking overhead vs. manual registration. Since most of our implementation relies on compile-time constructs with some runtime tracking, we expect this overhead to be negligible. This expectation is confirmed by the results in Figure 1, which show a stable trend for an increasing number of checkpointed views: at the extreme of 128 views, the overhead is less than 0.06 seconds, compared with the bookkeeping overhead of *VELOC* which is 0.01. seconds. Given that most applications need to checkpoint few but large data structures, we conclude that our bookkeeping overhead is negligible.

Next, we evaluate the *ping-pong* microbenchmark that was discussed in Section 5. This benchmark measures the reduction in checkpointing overhead of using the *const* tracking optimizations versus a baseline case that is not using them. The benchmark exhibits a 50% reduction in checkpointing size due to *const* tracking (i.e., only one out of two equally sized memory views is checkpointed). As observed in Figure 2, this translates in practice to an almost 30% speedup over the non-optimized version.

### 6.4 Results: HPC Mini-apps

In order to analyze the performance of our approach in a holistic manner, we performed several scaling experiments with the two scientific mini-apps mentioned in Section 6.2.

[3] Original implementation at `https://github.com/Mantevo/miniMD`
[4] `https://github.com/ECP-VeloC/VELOC`

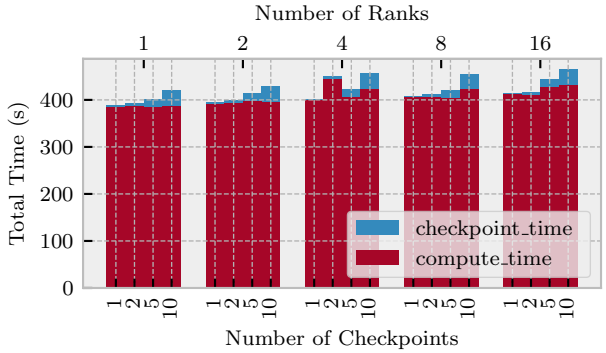


**Fig. 3.** *MiniMD*: Breakdown of checkpointing time vs. compute time for a increasing number of nodes and checkpoints in a weak scalability scenario over 1000 time steps.

Figure 3 shows a broad overview of the performance of our checkpointing scheme on the MiniMD molecular dynamics app with 13 million atoms and 1000 timesteps. We vary both the number of checkpoints executed during the entire simulation run and the number of ranks. MiniMD scales weakly, as we can see by the red bars in the figure. Even at a high number of ranks, the checkpoints do not become more expensive. Furthermore, a large number of checkpoints (one every 40 seconds) has very little impact on the total execution size for this problem.

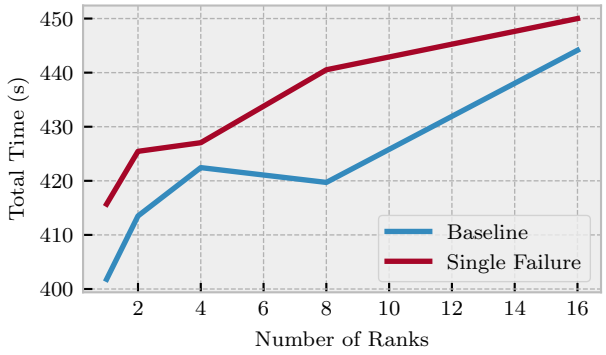


**Fig. 4.** *MiniMD*: Comparison of a baseline checkpoint every 200 iterations and a checkpoint/restart with a failure at iteration 401 in a weak scalability scenario.

Figure 4 gives an overview of performance with a checkpoint every 200 iterations with and without a failure.

In the *MiniMD* we also performed a brief evaluation of the reduction of code complexity for the user. The three main data structures used in the application contained a total of 65 views. Modifying the code for resilience required manual

tracking of only the three objects; this represents a significant decrease in code complexity compared to manually tracking all 65 views.

We performed another weak scaling study with the *HeatDis* heat solver mini-app. In this case, we varied the problem size (which directly corresponds to the size of the checkpoints). We compare a run that takes ten checkpoints (one every 100 iterations) without restart (Figure 5) and a run that does the same but also restarts from the third checkpoint (by simulating a failure at iteration 301). The results are depicted in Figure 6).

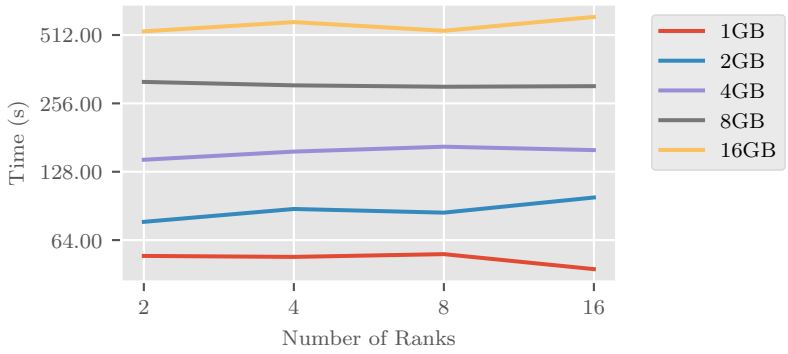


**Fig. 5.** *HeatDis*: Total runtime for an increasing number of ranks and checkpoint size per rank in a weak scalability scenario without restart (1000 iterations, checkpoint every 100 iterations).

As can be observed for both *MiniMD* and *HeatDis*, the evaluated scenarios exhibit minimal overhead due to checkpoint-restart, which is the result of combining optimized memory tracking with the asynchronous techniques introduced by *VELOC*. Based on the bookkeeping experiments discussed in Section 6.3, memory tracking has negligible overhead, while enabling automated checkpointing with minimal coding effort. Indeed, for our benchmarks and applications, a single lambda declaration (i.e., a single line of code) to wrap the code into a resilient execution context was enough to take advantage of the advanced checkpointing techniques exposed by VELOC.

From a qualitative perspective our approach solves three important challenges that manual checkpointing faces: (1) difficulty in identifying the critical data structures in complex codes that combine multiple execution contexts; (2) inefficient checkpoints that duplicate critical data structures or include data structures that are not critical; (3) deciding best moment to checkpoint the critical data structures.

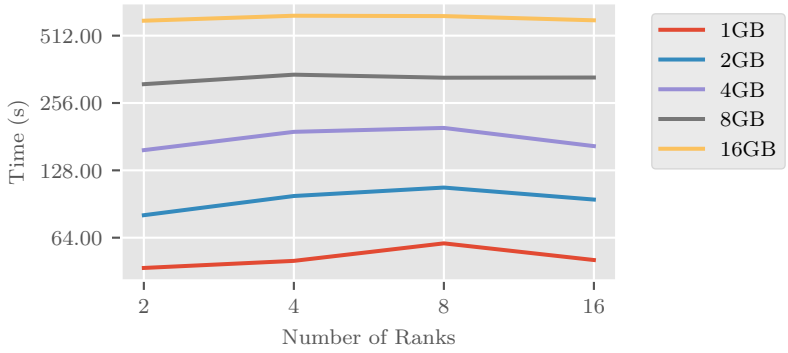


**Fig. 6.** *HeatDis*: Total runtime for an increasing number of ranks and checkpoint size per rank in a weak scalability scenario with restart (1000 iterations, checkpoint every 100 iterations, restart at iteration 301).

## 7 Conclusion and Future Work

Our contribution is designed to provide convenient and efficient checkpointing for performance portable HPC applications. Our work extends the performance portable abstraction to that of *resilience portability*, allowing existing code to be made resilient with minimal changes. Moreover, we use type information determined at compile time to analyze the usage of resilient data, permitting the use of optimization based on usage patterns. We demonstrate the performance results of our work on various mini-apps and microbenchmarks. We show that compared to manually invoking state-of-the art resilience backends, we introduce negligible overhead.

In the future we would like to extend the idea of compiler analysis. Although compile-time data-flow analysis using C++ lambda introspection provides a useful way to determine inputs and outputs of checkpoint regions, a compiler pass could enable more robust detection, including analyzing any side-effects from references to global variables or static objects. Furthermore, we would like to extend our approach to a greater variety of backends and applications in order to understand better any performance implications and usability issues.

## Acknowledgments

This material is based upon work supported by the U.S. Department of Energy (DOE), Office of Science, Office of Advanced Scientific Computing Research, under Contract DE-AC02-06CH11357. Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration (NNSA) under contract DE-NA0003525. This work was funded by NNSA's Advanced Simulation and Computing (ASC) Program. This paper describes objective technical results and analysis. Any

subjective views or opinions that might be expressed in the paper do not necessarily represent the views of the U.S. Department of Energy or the United States Government.

## References


1. Baird, M., Fensch, C., Scholz, S.B., Šinkarovs, A.: A lightweight approach to GPU resilience. In: European Conference on Parallel Processing. pp. 826–838. Springer (2018)
2. Bautista-Gomez, L., Tsuboi, S., Komatitsch, D., Cappello, F., Maruyama, N., Matsuoka, S.: FTI: High performance fault tolerance interface for hybrid systems. In: SC '11: The 2011 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 32:1–32:32. Seattle, USA (2011)
3. De Kruijf, M., Nomura, S., Sankaralingam, K.: Relax: An architectural framework for software recovery of hardware faults. In: ACM SIGARCH Computer Architecture News. vol. 38, pp. 497–508. ACM (2010)
4. Di Martino, C., Kalbarczyk, Z., Iyer, R.K., Baccanico, F., Fullop, J., Kramer, W.: Lessons learned from the analysis of system failures at petascale: The case of blue waters. In: 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks. pp. 610–621. IEEE (2014)
5. Di Martino, C., Kramer, W., Kalbarczyk, Z., Iyer, R.: Measuring and understanding extreme-scale application resilience: A field study of 5,000,000 HPC application runs. In: 2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks. pp. 25–36. IEEE (2015)
6. Docan, C., Parashar, M., Klasky, S.: Dataspaces: An interaction and coordination framework for coupled simulation workflows. Cluster Computing **15**(2), 163–181 (2012)
7. Duan, S., Subedi, P., Teranishi, K., Davis, P., Kolla, H., Gamell, M., Parashar, M.: Scalable data resilience for in-memory data staging. In: 2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS). pp. 105–115. IEEE (2018)
8. Edwards, H.C., Trott, C.R., Sunderland, D.: Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. Journal of Parallel and Distributed Computing **74**(12), 3202–3216 (2014)
9. Gamell, M., Katz, D.S., Kolla, H., Chen, J., Klasky, S., Parashar, M.: Exploring automatic, online failure recovery for scientific applications at extreme scales. In: SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 895–906. IEEE (2014)
10. Gamell, M., Katz, D.S., Teranishi, K., Heroux, M.A., Van der Wijngaart, R.F., Mattson, T.G., Parashar, M.: Evaluating online global recovery with fenix using application-aware in-memory checkpointing techniques. In: 2016 45th International Conference on Parallel Processing Workshops (ICPPW). pp. 346–355. IEEE (2016)
11. Gamell, M., Teranishi, K., Heroux, M.A., Mayo, J., Kolla, H., Chen, J., Parashar, M.: Local recovery and failure masking for stencil-based applications at extreme scales. In: SC'15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 1–12. IEEE (2015)
12. Gamell, M., Van der Wijngaart, R.F., Teranishi, K., Parashar, M.: Specification of Fenix MPI Fault Tolerance library version 1.0. Tech. rep., Technical Report SAND2016-9171, Sandia National Laboratories, Livermore, CA (2016)

13. Hornung, R.D., Keasler, J.A.: The RAJA portability layer: overview and status. Tech. rep., Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States) (2014)
14. Hukerikar, S., Engelmann, C.: Resilience design patterns: A structured approach to resilience at extreme scale. Tech. Rep. ORNL/TM-2016/767, Oak Ridge National Laboratory, Oak Ridge, TN, USA (Dec 2016)
15. Laguna, I., Richards, D.F., Gamblin, T., Schulz, M., de Supinski, B.R., Mohror, K., Pritchard, H.: Evaluating and extending user-level fault tolerance in MPI applications. The International Journal of High Performance Computing Applications **30**(3), 305–319 (2016)
16. Losada, N., Bosilca, G., Bouteiller, A., González, P., Martín, M.J.: Local rollback for resilient MPI applications with application-level checkpointing and message logging. Future Generation Computer Systems **91**, 450–464 (2019)
17. Martsinkevich, T., Subasi, O., Unsal, O., Cappello, F., Labarta, J.: Fault-tolerant protocol for hybrid task-parallel message-passing applications. In: 2015 IEEE International Conference on Cluster Computing. pp. 563–570. IEEE (2015)
18. Moody, A., Bronevetsky, G., Mohror, K., De Supinski, B.R.: Design, modeling, and evaluation of a scalable multi-level checkpointing system. In: SC'10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 1–11. IEEE (2010)
19. Nicolae, B., Moody, A., Gonsiorowski, E., Mohror, K., Cappello, F.: VeloC: Towards high performance adaptive asynchronous checkpointing at large scale (2019)
20. Silveira, A., Ávila, R.B., Barreto, M.E., Navaux, P.O.A.: DPC++: object-oriented programming applied to cluster computing. In: Arabnia, H.R. (ed.) Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA 2000, June 24-29, 2000, Las Vegas, Nevada, USA. CSREA Press (2000)
21. Subasi, O., Arias, J., Unsal, O., Labarta, J., Cristal, A.: Nanocheckpoints: A task-based asynchronous dataflow framework for efficient and scalable checkpoint/restart. In: 2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing. pp. 99–102. IEEE (2015)
22. Teranishi, K., Heroux, M.A.: Toward local failure local recovery resilience model using MPI-ULFM. In: Proceedings of the 21st European Mpi Users' Group Meeting. p. 51. ACM (2014)
23. Tseng, S.M., Nicolae, B., Bosilca, G., Jeannot, E., Cappello, F.: Towards portable online prediction of network utilization using MPI-level monitoring. In: EuroPar'19 : 25th International European Conference on Parallel and Distributed Systems. pp. 1–14. Goettingen, Germany (2019)
24. Van Der Wijngaart, R.I., Gamell, M.R.U., Teranishi, K., Valenzuela, E., Heroux, M.A., Parashaar, M.R.U.: Fenix; a portable flexible fault tolerance programming framework for MPI applications. Tech. rep., Sandia National Lab.(SNL-NM), Albuquerque, NM (United States) (2016)