# A Foundation for Automated Placement of Data

Douglas Otstott, Ming Zhao
*Arizona State University*
*Email: (dotstott,mingzhao)@asu.edu*

Sean Williams
*New Mexico Consortium*
*Email: swilliams@newmexicoconsortium.org*

Latchesar Ionkov, Michael Lang
*Los Alamos National Laboratory*
*(lionkov,mlang)@lanl.gov*

## Abstract

With the increasing complexity of memory and storage, it is important to automate the decision of how to assign data structures to memory and storage devices. On one hand, this requires developing models to reconcile application access patterns against the limited capacity of higher-performance devices. On the other, such a modeling task demands a set of primitives to build from, and a toolkit that implements those primitives in a robust, dynamic fashion. We focus on the latter problem, and to that end we present an interface that abstracts the physical layout of data from the application developer. This will allow developers focused on optimized data placement to use our abstracta as the basis for their implementation, while application developers will see a unified, scalable, and resilient memory environment.

## 1 Introduction

What was once a few distinct tiers is now a blurred range of memory and storage hardware with complicated tradeoffs in performance, capacity, and resilience. This trend is expected to continue, particularly with the push toward exascale, as the problems posed by memory and storage latency come into ever-sharper relief. This poses a serious problem for application developers: the decision of how to place data structures within a complex array of memory and storage devices is often subtle, and comparatively few of them are the kinds of "hero programmers" who will muddle through the ambiguity.

Meanwhile, there are efforts to dynamically optimize the placement of data structures. For example, our collaborators are looking at testing the performance of codes under different mappings of data to devices [1, 2]. For application developers, it should be easy to use these data placement optimizations. For runtime developers, the optimization of data placement requires the ability to easily run an application under different memory and storage configurations, so this problem is quite similar for them.

To satisfy runtime developers and application developers, we develop an infrastructure to support the mechanisms for the optimization of data placement paired with a simple API for application developers. We leveraged a variety of prior work to construct a runtime that, on one hand, only requires application developers to state their data in terms of a formal language, and on the other, provides hooks to allow the details of data structure placement to be specified to any level of detail. The runtime is specified in two parts: a node-local part that manages placement of node-local data, and a global-shared part that automatically reconciles data between nodes. Various implementation details rely on explicit versioning of data structures, which, besides making data semantics clearer, carries side benefits like asynchronous checkpointing and easy-to-use ghost cells[1].

Our primary focus with this work is on functionality and convenience, but we will also show results to the effect that, running on real codes, our framework has negligible effect on performance. Indeed, deploying asynchronous I/O into codes that didn't have it before, itself done with only minor change, drastically improves checkpoint performance. Nonetheless, our primary focus is on infrastructure supporting optimized data placement, so these results are presented primarily as a sanity check. The work presented in this paper doesn't include applying any specific automated data placement, development of cost models, or online or offline data reorganization.

[1]"Ghost cells" are cells along the boundaries between different subsets of a large data set. These are used because most grid-based numeric simulations rely on stencil-style operations, i.e., computing the next value in a cell based on the current value in neighboring cells. Along subgrid boundaries, some neighboring cells reside on different nodes, so these values are typically communicated as an explicit operation at time step boundaries.

## 2 Related Work

### 2.1 ADIOS

The Adaptable IO System (ADIOS) [3], developed at Oak Ridge National Laboratory, provides a simplified storage abstraction which implements a range of I/O transport methods used by scientific applications. Users provide an external XML file which outlines data elements, their types, and the intended method of processing. Currently, ADIOS only supports primitives and arrays, and must be recompiled whenever static values (like transport method) have been changed. This prevents the reimplementation of at least one application (VPIC) used during evaluation. ADIOS also includes a set of transformation functions which the developer can use to compress the dataset or reindex arrays when passing between C and Fortran. While similar to DRepl, which will be discussed in Section 4.1, it does not support an entire language to describe the range of possible transformations.

### 2.2 Legion

Legion [4, 5], a data-centric programming model developed by Stanford University, is a library and runtime for executing applications on distributed parallel systems. The authors explain in detail how the relationships between subsets of data can be leveraged to inform decisions about data placement and process scheduling. Essentially, at compile time, Legion maps each computation step to its dependencies, i.e., the input of one step is the output of another. Legion then associates each region of data produced by the application with a specific computational step. Legion exposes a simple API for developers to write their own mapper. However, this is explicitly for CPUs and memory only: Legion provides no interface for storage access outside of the usual POSIX functionality. Also, Legion does not implement optimized data placement, so the runtime cannot dynamically respond to performance.

### 2.3 DAOS

The Distributed Application Object Storage (DAOS) [6] layer provides an abstraction between the underlying hardware and any number of applications, or even more abstraction. DAOS itself exposes an object/container interface; the authors have already written an HDF5 interface for the top layer, and intend to replace that with a filesystem. Meanwhile, within DAOS, intelligent decisions are made which dictate where data are placed. DAOS also supports versioning and leverages copy-on-write technology to protect consistent versions. On its own, DAOS does not fully abstract the I/O stack: whichever process is interfacing directly with DAOS must explicitly refer to a storage device when it creates a storage pool. DAOS also does not track all application data; rather, it must wait until an explicit object write operation has been issued. Since it is largely event driven it, it lacks the ability to optimize and refine its data placement strategies during lulls in I/O.

## 3 Application Programming Interface for Data Sharing and Storage

Modeling the optimal placement of data structures on memory and storage devices, generally speaking, proceeds from scoring allocations on how their placement impacts the performance of an application. For example, one might run an application multiple times with the same input deck, and with a different data structure in higher-performance memory each time [1]. The different execution times, then, provide some insight into the relative sensitivity of an application's performance to each structure.

With a more simplistic approach to measuring and scoring data structures, one can do the required bookkeeping by hand. For example, if a code has a small number of data structures of interest, then one can plausibly go through a "static" testing protocol (i.e., a data structure is or isn't in higher-performance memory for an entire run) for a small number of configurations. This kind of approach generally isn't scalable, particularly if one wants to run temporally dynamic tests, i.e., ones in which data structures are migrated between backing devices at different times in the run.

Thus, more sophisticated modeling approaches would require a higher-level specification of one's testing protocols—that one be able to specify tests algorithmically—as well as a reasonable way of implementing one's results. This requires that one's data structures become "first class," which is to say, they need to be specified at the same level of abstraction as one's modeling. This extends not just to the declaration of one's data, but to the allocation and management: all of these things must be explicit in the programming environment.

In summary, we believe that a framework for easing the development and deployment of optimized data placement must provide an abstraction over the declaration, initial placement, and continued management of a code's important data structures.

### 3.1 Application Interface

We will discuss our general design in terms of an example simulation, shown in Figure 1, which is a simple heat transfer simulation. It has a 2D grid, which is broken

```
1   #define G(x, y) \
2     (y == -1 ? (ug ? ug[x+1] : 0) : (y == ymax ? (bg ? bg[x+1] : 0) : \
3     (x == -1 ? (lg ? lg[y] : 0) : (x == xmax ? (rg ? rg[y] : 0) : \
4     g[(x) * xrange + (y)]))))
5
6   #define AVG(a, b, c, d, e, f, g, h, i) \
7     0.28 * a + 0.14 * (b + c + d + e) + 0.04 * (f + g + h + i)
8
9   int length = 30, height = 30, nlength = 6, nheight = 5
10
11  int main() {
12    double *g, *ug = NULL, *bg = NULL, *lg = NULL, *rg = NULL;
13    int x, y, itr;
14
15    MPI_INIT();
16    MPI_Comm_Rank(MPI_COMM_WORLD, &rank);
17
18    if(rank == 0)
19      create_object("heatsim", "var grid [length,height] float64", 0);
20    MPI_Barrier(MPI_COMM_WORLD);
21
22    xrange = length / nlength;
23    yrange = height / nheight;
24    xmin = (rank % xrange) * yrange;
25    ymin = (rank / xrange) * yrange;
26
27    attach("heatsim", "{var grid[x:xrange,y:yrange] = grid[x+xmin,y+min
            ]}", 0, 0, &g);
28    if(xmin > 0)
29      attach("heatsim", "{var rank_yghost[x:1,y:yrange] = grid[x+xmin-1,
              y+ymin]}", 0, 1, &lg);
30    if(xmin + xrange < length)
31      attach("heatsim", "{var rank_yghost[x:1,y:yrange] = grid[x+xmin+
              xrange,y+ymin]}", 0, 1, &rg);
32    if(ymin > 0)
33      attach("heatsim", "{var rank_xghost[x:xrange+2,y:1] = grid[x+xmin
              -1,y+ymin-1]}", 0, 1, &ug);
34    if(ymin + xrange < height)
35      attach("heatsim", "{var rank_xghost[x:xrange+2,y:1] = grid[x+xmin
              -1,y+ymi+yrange]}", 0, 1, &bg);
36
37    // Initialize g (the simulation grid) with some values
38
39    for(itr = 0; itr < 100; itr++) {
40      for(x = 0; x < xrange; x++)
41        for(y = 0; y < yrange; y++)
42          g[x*xrange+y] =
43            AVG(G(x,y), G(x-1,y), G(x+1,y), G(x,y-1), G(x,y+1), G(x-1,y
                  -1), G(x+1,y-1), G(x-1,y+1), G(x+1,y+1));
44
45      publish("heatsim", g, itr);
46      if(lg) reattach("heatsim", lg, itr);
47      if(rg) reattach("heatsim", rg, itr);
48      if(ug) reattach("heatsim", ug, itr);
49      if(bg) reattach("heatsim", bg, itr);
50    }
51  }
```

Figure 1: A complete example code, showing the creation of an object (line 24), local data (line 32), ghost cells (lines 33-40), version updating (line 50), and ghost cell synchronization (lines 51-54). This is a simple, synthetic heat transfer simulation: at each time step, each cell is updated to be the weighted average of itself and its eight neighbors.

apart across several MPI ranks, and each cell is updated to a weighted average of itself and its eight neighbors. The boundary is treated as an infinite sink, i.e., all boundary values are always 0.

We begin with specifying the code's important data in terms of DRepl [7] statements. For example, on line 24,

```
"var grid [length,height] float64"
```

This is evaluated in terms of a namespace, `heatsim` in this example, to create a data object (i.e., one or more data structures):

```
create_object("heatsim",
  "var grid [length,height] float64", 0);
```

A data object is just a description, similar to the role filled by classes in object-oriented programming. They serve a role of coordinating definitions across processes and nodes, but data are actualized in terms of subsets, which we call *fragments*. This is a natural if somewhat obscure formulation, as most important data structures in distributed codes are shared across nodes, and making this local/global distinction explicit aids the rest of the design. Suffice it to say, a fragment is an actual data structure, an actual address range, while an object is the abstractum that fragments are subsets of. Fragments are backed by memory, while objects describe how fragments relate to each other.

For example, taking `xmin` and `ymin` as starting indexes for some computation on the object `grid` for a particular rank of a distributed application, then we define a fragment of `grid` on line 32 as,

```
attach("heatsim", "{var grid[x:xrange,y:yrange] =
  grid[x+xmin,y+min]}", 0, 0, &g);
```

The third argument, `0` in this case, is a version number. This is done so that we have a coherent way to describe shared data: data at the current version are still volatile, while data at previous versions are fixed and safe to read. More importantly, this provides a straightforward-but-mechanistic way to know when data transmission should occur, and when blocking must occur—which is important, if we're to credibly claim to provide a "complete" abstraction for simulation data.

For example, ghost cells along a left boundary can be defined, as on line 34,

```
attach("heatsim", "{var rank_yghost[x:1,y:yrange] =
  grid[x+xmin-1,y+ymin]}", 0, 1, &lg);
```

That this is a consumer of shared data is indicated by the fourth argument, which, when set to `1`, makes a fragment read-only. A producer of shared data, on the other hand, must publish it, on line 50:

```
publish("heatsim", g, itr);
```

A consuming `attach` call will block until the data it requests have been published by its producer, so in this sense, as long as the developer keeps the version numbers consistent between processes, a publish/subscribe model of data sharing is automatically implied. This is one of the most important patterns of distributed simulations: data for a time step is computed from local data plus ghost cells, then new ghost cell values are broadcast at time step boundaries. More generally, the pattern of local computation during a time step and global reconciliation between them accounts for nearly all distributed simulation data wrangling problems, which is covered by the publish/subscribe model. This shows how our framework is easy-to-use for typical scientific simulation tasks concerning data management.

Since attaching and publishing are explicit function calls, they provide hooks to carry out more specialized operations. For example, if one is testing the performance effect of having a particular data structure on higher-performance memory, then one could modify the simulation to take an object name as a command-line argument, and use the hooks within `attach` to handle the placement. So for this example, one can then run tests without recompiling the code—and, more importantly, a wrapper script could enumerate all the data structures.

Finally, an instance of automated data placement could be written as a separate generic library, and could then be parameterized for a particular simulation by an input deck. In this sense, this framework should make the optimization of data placement simpler.

## 3.2 Data Object Versions

Versions are used to provide a consistent view of an object without explicit global synchronization. Since there is no single data layout, data is produced and retrieved at fragment granularity, and since there might be many replicas of the same data, it is hard to establish what constitute the latest data and whether they are in a consistent state. The most popular approach currently is to encode the version of the data in the file(s) produced by the application and to ensure that all data are written by global communication between all producer processes. The effect of global synchronization on application performance will increase even further with the number of processing elements required for exascale, making the current solution prohibitively expensive.

Associating versions to the fragment data produced by each process avoids the global synchronization approach and allows independent execution progress. The name-server, as a global service, can keep track of the versions of all fragments and establish the latest version of the object as a whole. That is, using versioning makes the semantics of queries about data more explicit; it forces developers to confront the fact that data exist in potentially many states, a fact that is reified by a versioning system.

Additionally, the version attribute can be used by applications when sharing data between processes. For example, a process can wait until data that it needs for the next step, such as ghost cells/halo regions, is available, without explicitly knowing which other ranks produce them. Data maintenance services can wait until all data for an object are available before persisting it to non-volatile memory, or for a new version to become available before deleting an old one.

## 4 Runtime Framework

## 4.1 Framework Components

As mentioned in Section 1, our approach leverages several preexisting libraries, which we will briefly describe here.

First, DRepl [7] is a formal language and interpreter for describing data structures, and for specifying and evaluating transformations between data layouts. It supports the ordinary primitive types, as well as arrays and (non-self-referencing) structured types. The transformation mechanism allows one to declare a variable to be a subset of another; for example, an array can be a slice or stride of another array, and a structure can be a substructure of another. By using such a system, we can process variable declarations in potentially arbitrary ways, which provides the base mechanism for intervening on the placement of data.

Its ability to handle transformations provides further opportunity for optimized data placement. For example, certain subsets may be indicated as read-only in some contexts, as is the case with ghost cells. If a runtime is aware of the relative read and write performance of each device, it can inform data placement decisions using DRepl to perform the required transformations.

Second, the required bookkeeping—tracking names and the locations of their referents—is carried out with distributed key/value stores. We chose Hop [8], which is a framework for building such stores; this allows greater flexibility in the choice of features, to provide a better-tuned balance between consistency, scalability, and availability. In particular, Hop can support versioning of entries, which is being used for data object versions, described in 3.2.

Third, we employ a modified Linux kernel called TCASM [9, 10], which supports asynchronous memory sharing between processes via changes to the `msync` system call. TCASM leverages Linux's copy-on-write page tracking to reduce the memory overhead needed to share data and eliminate the need for any synchronization. Applications using TCASM do not need to even know about each other to work, and since memory is shared asynchronously, copies are far faster than user-space memory-sharing methods. This enables features such as asynchronous checkpointing and interprocess communication.

## 4.2 Infrastructure Services

We employ a distributed two-tier architecture: the lower tier acts as a memory allocator and storage manager for the node on which it is resident, while the higher tier provides a global view of the whole computing environment,

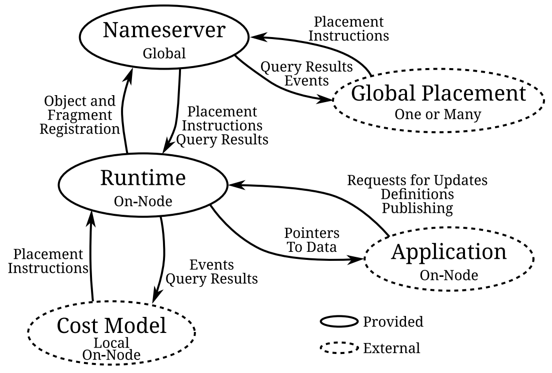

Figure 2: An architectural diagram of on-node runtimes and global services, as well as the "hooks" provided for local and global cost models for optimized data placement.

as well as the global storage resources. We will refer to these persistent services, respectively, as the runtime and the nameserver. The application then interacts with these services through the defined API.

The on-node runtime is the key element of our framework. It runs as a separate process and tracks all fragments located on the same node. It can interface with other on-node runtime instances over the local network and uses the nameserver to locate the runtime responsible for data fragments that are not locally available, e.g., when it needs to read data from a newly-created fragment. It also registers locally-produced fragments into the nameserver so they can be found by other runtime instances that need the data. The runtime is designed to interface with any available storage medium (e.g., traditional file systems, NVM, remote storage). It tracks all data fragments physically stored on media available to its host and provides hooks to other services for optimizing data placement, replication, and persistence.

We do not require applications to use our runtime to manage all data; for example, an application may allocate only certain data structures via our runtime, while using traditional system calls to manage some file I/O. However, to achieve the full range of functionality and portability we offer, applications should use the runtime for all significant allocations. In particular, any data that need to be persisted (either as output or checkpoint) or shared between processes should be managed by our runtime.

The upper tier, the nameserver, is responsible for tracking all allocations across all runtimes. The nameserver keeps track of all data objects (i.e., abstract containers), their fragments, and how many versions there are of each fragment. The nameserver allows creation of objects and fragments, and provides information on how to produce new fragment data from the preexisting fragments. It is aware of what objects and fragments are created as well as the locations of the fragments' data, but doesn't participate in the actual movement or transformation of data that may be required by reads or writes.

The nameserver is currently a single process, but the infrastructure supports a distributed implementation. It may live on a set of compute nodes, alongside runtimes, or it may reside on a distinct partition of nodes entirely. The nameserver maintains a database where it stores the metadata on each allocation across the whole system. Its primary function is to service queries from the runtimes about data managed by other runtimes.

## 4.3 Details of Runtime Operation

The operation of the nameserver is fairly straightforward: It is a key/value service implemented in Hop [8], which primarily serves to index fragments. This is important, because fragments can be arbitrarily diffused throughout a computing environment, i.e., a particular fragment could reside on any node in a cluster. If a process needs to access a particular fragment, it needs to know where to look. Thus, the nameserver implements typical key/-value operations: creation, update, query, and removal of objects and fragments. Its most important operation is the ability to specify a fragment which could, itself, be a union and intersection of several other fragments, and get back the locations of the source fragments and the DRepl transformation rules that convert the source fragments into the desired fragment. The nameserver therefore does most of the background work required to assemble a fragment from a potentially-heterogeneous collection of source fragments from its providers (i.e., the processes or nodes that computed the data), a task of which computing ghost cells and collating data for output to a single file are two simple examples.

The lion's share of this work resides in our runtime, which manages data that are local to a node. While the nameserver maintains and provides information—for example, about the steps required to create a fragment from other fragments—it is the runtime that actually carries out the work. Thus, on one hand, it communicates with the nameserver to register its objects and fragments and to find others' objects and fragments, and on the other, it communicates with other runtimes in order to retrieve and transmit actual data to materialize fragments to service applications' needs.

Besides the three operations implied in Figure 1—creating, attaching, and publishing fragments—there are two more important functions of the runtime. First, runtimes can detach a fragment, which will delete then fragment when there are no more references to the data. Second, a runtime can receive DRepl transformation rules, in which case it will apply those rules to a copy of the involved fragments and return the data in a compact intermediate format. The second function is used by runtimes on other nodes for materializing a fragment locally.

TCASM is used for efficient and consistent sharing of fragments' data between a runtime and its local applications. By versioning data, with the rules that prior versions are read-only and data are only shared after they have been rendered read-only, it is always safe to transmit data asynchronously. TCASM provides the machinery for this operation: publishing a fragment, and therefore incrementing its version, prompts the kernel to copy the relevant page on the next write, so that the existing page contains the previous version.

This all comes together to implement a persistent framework of services that the application interacts with to optimize and store application data. As an illustrative example: the application creates a fragment, this fragment is handed to the on-node runtime, the runtime registers this fragment with the global nameserver, and the on-node data-mover of the runtime queries a cost model to determine placement the data on the appropriate storage media, as depicted in Figure 2.

## 5 Results

As the system is not intended to adversely affect the performance of applications, results are focused on demonstrating the lack of overhead incurred. In addition to testing the system separately, we selected two use cases that are representative of a range of scientific applications for the system, to demonstrate the range of functionality and ease of development required for their implementation. We used two preexisting codes: SNAP [11], a proxy application that represents a deterministic neutral-particle transport simulation written in modern FORTRAN, and VPIC [12, 13, 14], a general-purpose particle-in-cell simulation for modeling kinetic plasmas in one, two, or three spatial dimensions written in C++. For all experiments, the setup includes running a single runtime on each node as a separate process, as well as running a single instance of the nameserver on one of the nodes.

| **Name** | Processor | Cores | Network | Nodes | DRAM |
|---|---|---|---|---|---|
| **Memdeep** | Intel E5-2660 | 20 | Ethernet | 1 | 64 GB |
| **CADDY** | Intel E5-2670 | 16 | IB-QDR | 150 | 32 GB |

Table 1: Experimental environments

The first experiment quantifies the time required to invoke the key functions: `create_object`, `attach`, and `publish`. To this end, we wrote a simple benchmark which first spawns a number of process, indicated by the x-axis on Figure 3. Processes are packed 16 per node, and each simultaneously creates 1,000 objects, recording the elapsed time. Since objects are the main meta-

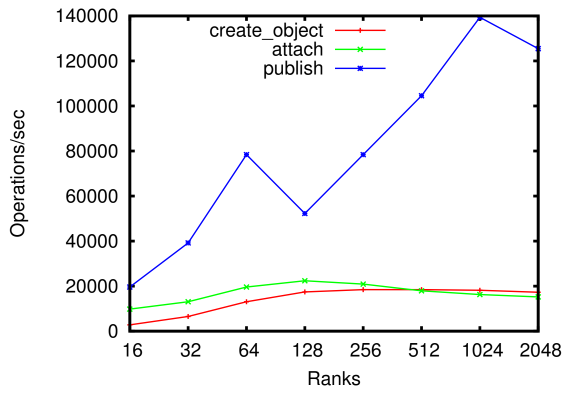

Figure 3: Average return time of select runtime operations: `create_obj` defines the global dataset for this application, i.e. the object; `attach` defines and initializes local variables (referred to as fragments) as subsets of the global dataset; finally the application uses `publish` to notify the runtime when it has finished a new version of an previously attached fragment.

data structure indexed by the nameserver, all entries compete for access, resulting in the highest contention of all operations. After the creation of the objects was concluded, each of the processes then performed `attach` 1,000 times for each of the 1,000 objects they had just created. Creating and attaching to fragments requires updating the object metadata structure, though only the object of interest needs to be locked, resulting in less, but still some, contention. However, it also requires memory allocation at the node level. In the end, these qualities balance out, resulting in roughly the same performance: 20,000 operations per second for both `create_object` and `attach`. Finally, all the attached fragments are iterated over, in order to populate the dataset with dirty pages. All processes block until they have finished writing and call `publish`, simultaneously. Publishing a new fragment version requires both a call to `msync` and an update to the metadata stored in both the runtimes and the nameservers. However, the majority of updates are made to a fragment, which is quite fast. Again, due the granularity of the update, publishing fragments requires the least overhead, performing operations at nearly 140,000 per second. The results of this experiment are shown in Figure 3. This rate is adequate, which will be shown in the application results: the overhead did not impact performance of real applications. VPIC, specifically, was under 20K calls to `attach` per second. Further optimization of these services can be addressed should new application requirements demand it.

To measure the memory overhead incurred by the runtime on a known mini-application, a single node, Memdeep (see Table 1), was used to run a series of MPI jobs with 20 ranks, using about 50% of its DRAM.

| **SNAP Run** | Time(s) |
|---|---|
| **Original** | 850 |
| **RT/NS** | 925 |

Table 2: SNAP's runtime compared SNAP implemented to use the runtime to allocated and publish application data on a single node, 16 processes and no file I/O.

Ranks were pinned to single cores and allowed to spawn two OpenMP threads, to make use of the Xeon's hyper-threading. The results of this experiment are recorded in Table 2. In total, the system incurs about 8% performance overhead, or 72 seconds. Based on prior experiments with TCASM [9], the overhead is likely due to the copying-on-write mechanism. In the worst-case case scenario, an application may write to each page of its allocation, effectively doubling its memory footprint. However, as TCASM allows developers to define data of interest, it is unlikely the entire application would be allocated via TCASM's allocator. Depending on the use case, the amount memory handled by TCASM can vary significantly. It is also noteworthy that the performance of TCASM is heavily affected by the access patterns of the applications, primarily the percentage of pages touched between calls to `msync`.

Since the primary focus of the system is memory and storage management, a use case was selected to highlight its ability to persist coherent datasets. First, SNAP was modified to use the runtimes to allocate all data necessary to perform a checkpoint/restart, and publish versions of this data at regular intervals using the `time_step` variable as the version number. Second, the original SNAP code was modified to use the collective MPI_IO read and write functionality to checkpoint and restart the same set of data. Intervals of 4 and 8 time steps were chosen to write the checkpoint, performed by a collective MPI_IO write operation in the latter case and a coordinated effort by the runtimes and the nameserver in the former. Full restarts were performed as sanity checks for both implementations.

Experiments were conducted with both implementations deployed on Memdeep. Since it includes a SATA SSD and an NVMe device, the experiment was repeated using each medium as the final destination of the persistent datasets. The results of these experiments are shown in Figure 4.

The greatest performance increase was achieved when persisting checkpoints on a spinning hard disk. The runtime/nameserver (RT/NS) implementation of SNAP's checkpoint finished 11 minutes faster than the MPI_IO inline checkpoint. By switching to the SATA SSD, the runtime saw its performance remain steady, while the application's total execution time improved by 8.5 minutes. The final experiment, done on the NVMe device, saw further improvement for application checkpointing, but only about 40 second while the runtime's checkpoint was unchanged.

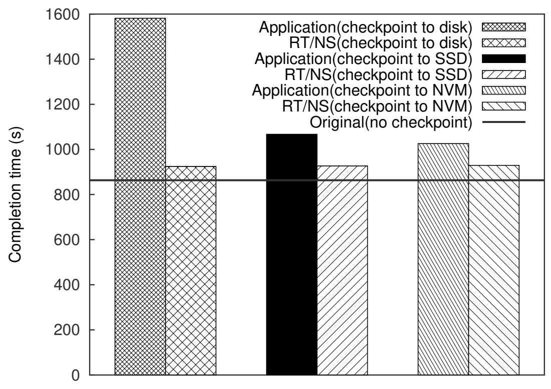

Figure 4: SNAP Three Ways: The original out-of-box release was used as a baseline to compare the modified implementations; the Application checkpoint implementation has all the I/O operations contained with one function which uses MPI_IO to coordinate the collective file write; and RT/NS implementation uses the on-node runtime to allocate and publish its dataset offloading the actual file I/O to the runtime process.

We see our system performs at the same level regardless of the medium, this is mainly due to TCASM's contribution to the system: asynchronous I/O, which allows the application to continue uninterrupted while the runtime performs the checkpoint. Meanwhile, the traditional in-line checkpoint function requires a blocking write, so that no sensitive data may be modified before the checkpoint is finished.

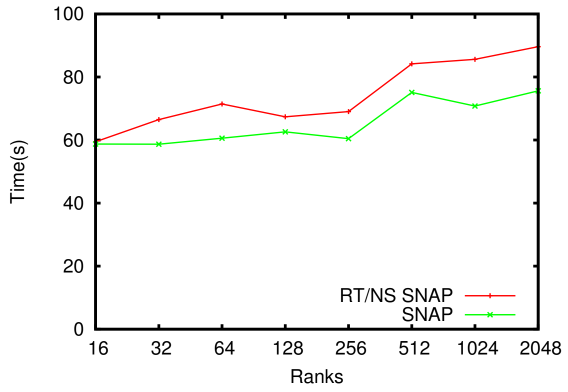

Figure 5: Overhead imparted by the runtime: The execution times of the original SNAP are compared to those of SNAP ported to RT/NS, for all available input decks.

Next, the generic, runtime, and MPI_IO implementations were deployed at scale using 128 compute nodes

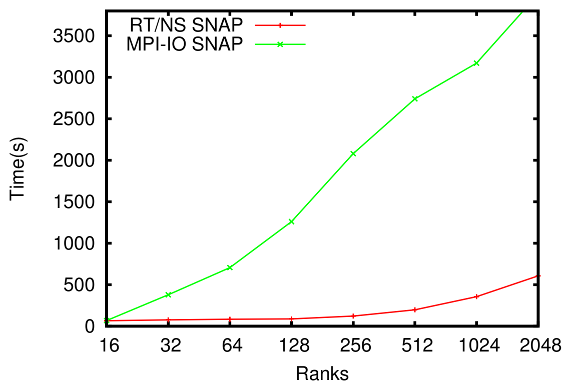

Figure 6: The MPI_IO library was used to implement a collective checkpoint function tuned to the Lustre filesystem's configuration, which is compared to the RT/NS SNAP with the same set of input and configuration as in Figure 5 with the persist feature has been turned on.

from a Linux cluster named Caddy (see Table 1). Caddy uses the SLURM resource manager to allocate nodes and invoke MPI jobs. All compute nodes are attached to a parallel Lustre filesystem, which was used for storing all checkpoints. For this reason, the MPI_IO implementation was tuned to use the `lustre fs` MPI component. Packing MPI jobs up 16 to a node (1 rank per core), the problem size was incrementally increased to 2048 cores across 128 nodes. Figure 5 compares the execution times of unmodified SNAP against the one using the distributed runtime and nameserver in order to demonstrate the system's performance overhead at scale. As in the single node experiments, overhead incurred by the use of the runtimes and nameserver to allocate and track data was minimal. In the worst case (2048 nodes), the use of our system only added about 40 seconds (or about 11%) to the total execution time.

Figure 6 shows the same experiments, except comparing the MPI_IO SNAP to the RT/NS SNAP implementation with the persist feature activated. For fairness, the times reported include any extra time the runtimes needed to finish the checkpoints asynchronously if SNAP finished while the runtimes were waiting on the write operation to complete.

Due to TCASM's asynchronous consistency guarantees, the runtime/nameserver implementation of checkpointing in SNAP, from the application developer's perspective, amounts to a memory update operation (`msync`) and a local interprocess message (`publish`), allowing the computational work of all 2048 processes proceed immediately. Even with MPI_IO's Lustre component tuned to Caddy's Lustre filesystem, the flushing, process synchronization, and blocking required to write the checkpoint incurs significant overhead. During the larger experiment (2048 ranks across 128 nodes), the difference in execution time reported by the MPI_IO implementation was 6.5 times that of the version which leveraged the runtime and nameserver for persistence (over one hour vs. roughly 8 minutes).

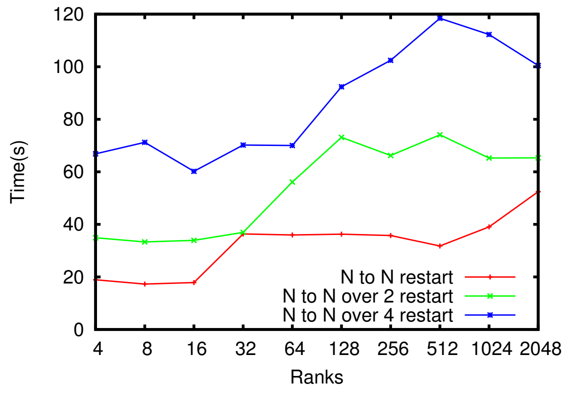

Figure 7: Each experiment begins with N ranks of RT/NS SNAP, which all exit after they publish each of their fragments at the end of the first iteration. Next, N SNAP ranks restart, attaching to the previously published fragments. The red line indicates the time SNAP took to restart only. For the rest of the experiment, the number of MPI processes was halved and quartered to draw the green and blue lines respectively. The resulting graph shows how long is required to repartition SNAP's datasets, enabling it to restart with fewer nodes than it originally had at start up.

In keeping with our theme for a foundation for automated data placement, we designed an experiment that would allow an application to checkpoint on N ranks and then restart on M ranks to show the runtime's ability, to abstract away the underling details of the supercomputer.

The experiment leverages DRepl's ability to transform datasets without explicit intervention to restart a simulation using a different system configuration. Consider a compute cluster with N cores, running an experiment with 1 rank per core. At any point during an experiment's execution, some number of nodes could fail, leaving fewer than N cores available. The system allows the original dataset (partitioned for N ranks) to be repartitioned to the remaining cores. SNAP is a particularly useful application to demonstrate this, as its input deck defines all its global structures, which SNAP divides among processes at allocation time. Our SNAP implementation uses the runtime to allocate data structures, first defining the global arrays during `create_object` and then defining subsets of each global array within each process. Thus no additional modifications were required to restart with fewer or more processes than originally indicated, even if data from some nodes ends up on different nodes than where it was originally allocated.

Figure 7 shows the explicit data allocation costs for

N ranks, repartitioned to N, N/2, and N/4 ranks. Note that N-to-N partitioning requires runtimes to register new data with the nameserver and construct allocations to the specifications provided by DRepl. In each experiment, the process binding scheme was reorganized so that no process was assigned to the node where its allocations were already stored, requiring explicit internode transfers to build the datasets.

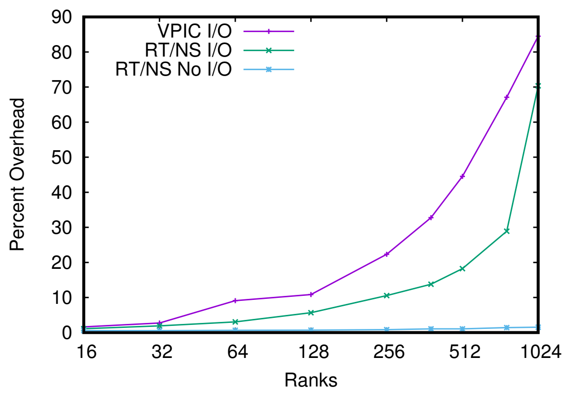

Figure 8: Unmodified VPIC comes with its own I/O functionality which it uses to write its output and checkpoints to persistent storage. We compare VPIC's native performance (VPIC I/O) against VPIC implemented to use the runtime and nameserver with and without persistence activated (RT/NS No I/O and RT/NS I/O respectively)

Finally, in order to demonstrate the system's efficacy with another modern application, VPIC's existing I/O operations were replaced with calls to the node local runtime API. VPIC has flexible support for saving its data at custom intervals. Starting from an existing VPIC input deck, we added support for weak scaling and the ability to use the local runtimes for data allocation and management.

Figure 8 depicts the relative performance of the new VPIC implementation. "VPIC I/O" refers to the original VPIC implementation with the existing I/O functionality activated in the input deck. "RT/NS" refers to the VPIC implementation which uses the runtimes to allocate data but no I/O is issued. "RT/NS I/O" refers to the same runtime data allocation implementation but file I/O is issued through the runtime/nameserver interface. The y-axis represents the overhead of the experiments as compared to running VPIC with the same input parameters, but without any I/O being performed. As with previous applications, the overhead of using the runtime for just memory allocations and fragment publishing is negligible. Adding I/O to the runtime increases the overhead to about 75% in the largest experiment, while the existing VPIC I/O functions nearly doubled the runtime of the original. Our modifications perform better than the default I/O, mostly due to the runtime's support for asynchronous I/O.

## 6 Future Work

As we have primarily described a middle layer between applications and optimized data placement, we are pursuing two directions of future work. On one side, to show the applicability of this interface to real-world simulations, we are working on modifying the Energy Exascale Earth System Model (E3SM), formerly known as Accelerated Climate Modeling for Energy (ACME), to use our code. This is the Department of Energy's next-generation coupled climate science simulation which, from the data perspective, is based on an unstructured (Voronoi) mesh. Besides being important to climate science, its multiscale mesh presents unique challenges for a storage abstraction.

On the other side, we are continuing our collaboration with computer scientists studying memory hierarchy performance and data migration. They are currently engaged in prototypical work, but once they feel they have refined their methods enough for deployment on real codes, we will be prepared to support those efforts. Our goal is to converge these two sides, to ultimately run relevant, large-scale simulations with their memory and storage subsystems automated, with the help of our framework.

## 7 Conclusion

We have described an interface between applications and data structure placement decision processes, the most prominent of which are based on memory hierarchy and storage performance trade-offs. On the application development side, this is based on "lifting" data declarations to runtime, via an interpreted data description language, and high-level functions which abstract the typical scientific simulation data flow, and provide hooks into the allocation and transmission functions.

We showed that, for developers, the resulting API is easy to fit into an existing code, and is reasonably agnostic to the type of code and the structuring of the data. Furthermore, because our runtime automatically makes use of asynchronous I/O, it is easy to, for example, reduce synchronization overhead for codes that need to exchange boundary information between time steps. We also showed that this convenience and flexibility does not meaningfully impact the performance of modified codes, and can in fact drastically improve performance by bringing asynchronous I/O to codes that didn't have it before without any extra development effort.

## References

[1] T. D. Doudali and A. Gavrilovska, "Comerge: Toward efficient data placement in shared heterogeneous memory systems," in *Proceedings of the International Symposium on Memory Systems*, ser. MEMSYS '17. New York, NY, USA: ACM, 2017, pp. 251–261. [Online]. Available: http://doi.acm.org/10.1145/3132402.3132418

[2] P. Fernando, S. Kannan, A. Gavrilovska, and K. Schwan, "Phoenix: Memory speed hpc i/o with nvm," in *High Performance Computing (HiPC), 2016 IEEE 23rd International Conference on*. IEEE, 2016, pp. 121–131.

[3] J. F. Lofstead, S. Klasky, K. Schwan, N. Podhorszki, and C. Jin, "Flexible io and integration for scientific codes through the adaptable io system (adios)," in *Proceedings of the 6th international workshop on Challenges of large applications in distributed environments*. ACM, 2008, pp. 15–24.

[4] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: Expressing locality and independence with logical regions," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012, pp. 66:1–66:11. [Online]. Available: http://dl.acm.org/citation.cfm?id=2388996.2389086

[5] S. Treichler, M. Bauer, and A. Aiken, "Language support for dynamic, hierarchical data partitioning," *SIGPLAN Not.*, vol. 48, no. 10, pp. 495–514, Oct. 2013. [Online]. Available: http://doi.acm.org/10.1145/2544173.2509545

[6] J. Lofstead, I. Jimenez, C. Maltzahn, Q. Koziol, J. Bent, and E. Barton, "Daos and friends: a proposal for an exascale storage system," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, 2016, p. 50.

[7] L. Ionkov, M. Lang, and C. Maltzahn, "Drepl: Optimizing access to application data for analysis and visualization," in *2013 IEEE 29th Symposium on Mass Storage Systems and Technologies (MSST)*, May 2013, pp. 1–11.

[8] L. Ionkov and M. Lang, *Hop: Elastic Consistency for Exascale Data Stores*. Cham: Springer International Publishing, 2015, pp. 291–306. [Online]. Available: https://doi.org/10.1007/978-3-319-20119-1_22

[9] D. Otstott, N. Evans, L. Ionkov, M. Zhao, and M. Lang, "Enabling composite applications through an asynchronous shared memory interface," in *Big Data (Big Data), 2014 IEEE International Conference on*. IEEE, 2014, pp. 219–224.

[10] H. Akkan, L. Ionkov, and M. Lang, "Transparently consistent asynchronous shared memory," in *Proceedings of the 3rd International Workshop on Runtime and Operating Systems for Supercomputers*, ser. ROSS '13. New York, NY, USA: ACM, 2013, pp. 6:1–6:6. [Online]. Available: http://doi.acm.org/10.1145/2491661.2481431

[11] J. Zerr and R. Baker, "Snap: Sn (discrete ordinates) application proxy - proxy description," 2013. [Online]. Available: https://github.com/losalamos/SNAP

[12] K. J. Bowers, B. J. Albright, L. Yin, W. Daughton, V. Roytershteyn, B. Bergen, and T. J. T. Kwan, "Advances in petascale kinetic plasma simulation with vpic and roadrunner," *Journal of Physics: Conference Series*, vol. 180, no. 1, p. 012055, 2009. [Online]. Available: http://stacks.iop.org/1742-6596/180/i=1/a=012055

[13] K. J. Bowers, B. J. Albright, B. Bergen, L. Yin, K. J. Barker, and D. J. Kerbyson, "0.374 Pflop/s Trillion-particle Kinetic Modeling of Laser Plasma Interaction on Roadrunner," in *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, ser. SC '08. Piscataway, NJ, USA: IEEE Press, 2008, pp. 63:1–63:11. [Online]. Available: http://dl.acm.org/citation.cfm?id=1413370.1413435

[14] K. J. Bowers, B. J. Albright, L. Yin, B. Bergen, and T. J. T. Kwan, "Ultrahigh performance three-dimensional electromagnetic relativistic kinetic plasma simulation," *Physics of Plasmas*, vol. 15, no. 5, p. 055703, 2008. [Online]. Available: http://dx.doi.org/10.1063/1.2840133