NVMalloc: Exposing an Aggregate SSD Store as a Memory Partition in Extreme-Scale Machines

Chao Wang[1], Sudharshan S. Vazhkudai[1], Xiaosong Ma[1,2], Fei Meng[2], Youngjae Kim[1], and Christian Engelmann[1]

[1]Oak Ridge National Laboratory, [2]North Carolina State University
{wangcn, vazhkudaiss, kimy1, engelmannc}@ornl.gov, fmeng@ncsu.edu, ma@cs.ncsu.edu

***Abstract*—DRAM is a precious resource in extreme-scale machines and is increasingly becoming scarce, mainly due to the growing number of cores per node. On future multi-petaflop and exaflop machines, the memory pressure is likely to be so severe that we need to rethink our memory usage models. Fortunately, the advent of non-volatile memory (NVM) offers a unique opportunity in this space. Current NVM offerings possess several desirable properties, such as low cost and power efficiency, but suffer from high latency and lifetime issues. We need rich techniques to be able to use them alongside DRAM.**

In this paper, we propose a novel approach for exploiting NVM as a secondary memory partition so that applications can explicitly allocate and manipulate memory regions therein. More specifically, we propose an *NVMalloc* library with a suite of services that enables applications to access a distributed NVM storage system. We have devised ways within NVMalloc so that the storage system, built from compute node-local NVM devices, can be accessed in a byte-addressable fashion using the memory mapped I/O interface. Our approach has the potential to re-energize out-of-core computations on large-scale machines by having applications allocate certain variables through NVMalloc, thereby increasing the overall memory capacity available. Our evaluation on a 128-core cluster shows that NVMalloc enables applications to compute problem sizes larger than the physical memory in a cost-effective manner. It can bring more performance/efficiency gain with increased computation time between NVM memory accesses or increased data access locality. In addition, our results suggest that while NVMalloc enables transparent access to NVM-resident variables, the explicit control it provides is crucial to optimize application performance.

I. Introduction

In the post-petascale era, scientific applications running on leadership-class supercomputers are ever more hungry for main memory. "Hero" jobs, running on O(100,000) cores, consume O(100TB) of memory for their processing. For example, a 100,000 core run of the GTS fusion application [29] on the Jaguar machine (No. 3 on the current Top500 [26] list) at Oak Ridge National Laboratory, consumes 200 TB of memory (at 2 GB/core.) As we approach several hundred petaflops or an exaflop, the memory requirements of applications are only likely to get more intense.

DRAM is an expensive resource in the HPC landscape and its provisioning consumes a significant portion of the multi-million dollar supercomputer budget. While it may seem that large-scale machines have a lot of memory, often to the tune of hundreds of TBs (e.g., 360TB on Jaguar), problem sizes attacked by modern HPC applications have been growing fast as well. Further, the memory-to-FLOP ratio has been steadily declining, from 0.85 for the No. 1 machine on Top500 in 1997 to 0.01 for the projected exaflop machine in 2018 [26], [19]. For example, the 2.5 petaflop Tianhe-1A (current No. 2 on Top500) has around 229 TB of DRAM, with a memory-to-FLOP ratio of 0.08. In addition, DRAM is a significant contributor to the supercomputer power budget. The shrinking memory/FLOP ratio can also be attributed to the desire to cap the power budget for next generation machines. Applications face the prospect of running wider to account for the ever shrinking memory/node, thereby incurring increased communication costs and, worse yet, increased usage of supercomputer allocation, a precious commodity mostly obtained through rigorous peer review.

The advent of non-volatile memory (NVM) devices, such as solid state disks (SSDs) offers a tremendous opportunity in such a setting. It is a well-known fact that flash technology is already serving to bridge the performance gap between DRAM and disk both in enterprise [16], [22], [5], [9] as well as HPC [11], [21] domains. To this end, today's large-scale HPC machines are being equipped with SSDs on the compute nodes. For example, the No. 5 machine in Top500 (Tsubame2) [26] has around 173 TB of total node-local SSD storage. Similarly, Gordon [17] at SDSC is an SSD-based data intensive cluster. We argue that NVM can also play a significant role in extending memory capacity in extreme-scale machines.

Flash technologies possess several features desired in DRAM, including low cost, high power efficiency, and high capacity. On the flip side, there are several other factors that limit their use as a substitute for main memory, such as high latency, access granularity, and limited lifetime. While the holy grail of "universal memory" (UM) that can replace both DRAM and NVM is still elusive [14], Mogul et al. propose a hybrid memory scheme that seamlessly combines NVM and DRAM with operating system support [15]. The premise here is that in the event it is desirable to replace DRAM with UM, the features of UM will require OS support. Even in

Table I
DEVICE CHARACTERISTICS. DEVICE PRICES ARE BASED ON CURRENT MARKET VALUES (OCTOBER 2011).

Device	Type	Interface	Read	Write	Latency	Cap. (GB)	Cost ($)
Intel X25-E [1]	SLC	SATA	250MB/s	170MB/s	75us	32GB	$589
Fusion IO ioDrive Duo [8]	MLC	PCIe	1.5GB/s	1.0GB/s	<30us	640GB	$15,378
OCZ RevoDrive [18]	MLC	PCIe	540MB/s	480MB/s	-	240GB	$531
Memory (DDR3-1600)	SDRAM	DIMM	12.8GB/s	12.8GB/s	10-14ns	16GB	<$150

commodity and enterprise-class systems, such an approach is still a long-term vision, let alone in the HPC landscape, where DRAM is integral to application performance and scalability.

Consequently, at the other end of the spectrum, we are faced with the following approaches for the use of NVM towards memory extension. First, the availability of node-local NVM makes it feasible to re-enable virtual memory on the compute node OS on extreme-scale machines (e.g., swapping to NVM). Traditionally, the OS on the tens of thousands of compute nodes on these systems has swapping turned off due to the absence of node-local disks. Since hard disks are particularly failure-prone, large-scale machines are usually not equipped with node-local scratch disks and are instead provided with high-speed central scratch parallel file systems. NVM offers several desirable features when compared to disks, such as superior throughput, lower access latency, and higher reliability due to the lack of mechanical moving parts, making it much more likely to be adopted as node-local storage. Although the access latency of a traditional NVM device (e.g., SATA-based SSD) is several orders of magnitude higher than DRAM, interfaces such as PCIe (e.g., FusionIO and OCZ flash cards) offer much lower latency (Table I.) Recent efforts [23], [12], [10], [20] have adopted the integration of a flash store with the Linux commodity OS virtual memory system as a means to extend memory capacity. Similar strategies can potentially be adopted for the optimized OS kernels running on HPC machine's compute nodes (e.g., Compute Node Linux).

Alternatively, a novel angle would be to expose the node-local NVM explicitly as a secondary, but slower memory partition for applications. While it is more complicated than virtual memory integration of NVM, this approach can offer better performance and a greater degree of control to applications, in allowing them to explicitly manage the NVM and dictate data placement. For instance, applications could potentially use the memory partition for operations that exploit the inherent device strengths, e.g., by allocating *"write-once-read-many"* variables onto the NVM. In particular, this usage model would be extremely beneficial for applications that conduct *"out-of-core"* computation.

The POSIX `mmap()` interface offers a viable way to map files or devices into memory. Using such an interface, sophisticated management structures can be built for the efficient use of node-local NVM. However, a realistic deployment scenario of NVM in future leadership machines makes this a non-trivial problem. For example, we noted earlier that PCIe connected SSDs offer lower latency and higher throughput than a typical SATA connected SSD (Table I). However, they are also expensive. Currently, a high-end Fusion I/O PCIe MLC flash card (io Drive Duo) at 640 GB is priced around $15K and offers around 1.1-1.5 GB/s read/write throughput, which is still at least 8.53 times lower than DRAM rates (Table I). Other PCIe offerings are cheaper, but provide further lower throughput. While NVM prices are continuing to reduce, the scale of current and future supercomputers makes it prohibitively expensive for each and every compute node to be equipped with an NVM device. For example, the Jaguar machine has 18,000+ compute nodes and future 100-300 petaflop and 1 exaflop machines are expected to host O(100,000) and O(1 Million) compute nodes, respectively. It is more than likely that only a subset of the nodes will be equipped with such devices (e.g., perhaps a partition of special "fat" nodes.) Thus, we need techniques so that clients can access such a partition of NVM-equipped nodes as a memory device.

In this paper, we have developed methods to expose a subset of NVM nodes as a secondary memory partition, from which applications can explicitly allocate memory regions and operate on them. Our key contributions are as follows:

NVMalloc: We have developed an NVMalloc library, comprising of a suite of services so that client applications can explicitly allocate and manipulate memory regions from a distributed NVM store. The NVMalloc library exploits the memory-mapped I/O interface to access local or remote NVM resources in a seamless fashion. In addition, the library provides an elegant approach to checkpoint both DRAM as well as NVM-allocated variables in an elegant, transparent manner. To the best of our knowledge, our work is the first in its attempt to enable memory-mapped accesses to an aggregate NVM store.

Revitalize Out-of-core Computation: Our solutions have the potential to re-vitalize out-of-core computation on large-scale machines with many-core processors. Applications can compute problems at a scale much larger than what the physical memory allows. Our approach presents a novel way to exploit the collective potential of node-local NVM and brings it bear on applications and machines faced with the

DRAM pressure.

Byte-addressability to Block Store Mapping: We have developed ways so that a block storage system can be accessed in a byte-addressable fashion. NVMalloc bridges the granularity gap between byte-by-byte accesses and large chunk-based aggregate distributed store through additional layers of data caches.

Evaluation: We have evaluated NVMalloc on a 128-core cluster with node-local SSDs using resource-intensive kernels such as matrix multiplication and sorting. Our results suggest the following: out-of-core execution using NVMalloc is a very viable solution (53.75% execution time improvement for matrix multiplication); smart data access patterns that can exploit NVMalloc's caching and SSD traits can achieve better performance; bridging `mmap`'s byte accesses against the block store is critical to the success of NVMalloc and the library can indeed enable efficient computations on problem sizes larger than the DRAM size.

II. Background: Aggregate NVM Store

In our prior work [11], [21], we have built an aggregate NVM storage system (using SSDs) from a subset of compute nodes and have demonstrated that such a storage system can be presented as an I/O impedance matching device between applications and the parallel file system (PFS) in extreme-scale systems. The aggregate NVM storage architecture is similar to our prior efforts on node-local disk aggregation (e.g., FreeLoader [27], stdchk [2].) In our design, compute nodes (or a subset of them) run a benefactor process that contributes the node-local NVM (or a partition of it) to a manager process that manages the aggregated NVM space to present a collective intermediate storage system to clients. The manager carries out tasks such as benefactor status monitoring, space allocation, data striping, and data chunk mapping. The aggregate NVM storage is made available to clients via a transparent file system mount point, (e.g., `/mnt/AggregateNVM`) using FUSE. A client that writes data to the mount point will be redirected to the aggregate SSD storage, without requiring any other code modification.

On a large-scale system, such an aggregate NVM store can be used as a checkpointing device or a staging store for large output/input data (as shown in our prior work [11], [21].) Our work has shown that when the NVM devices are distributed across a set of system nodes, aggregation and access through a file system mount point offers an elegant abstraction to transparently access them from the numerous compute nodes. This decouples the placement of NVM from the compute nodes and allows for sharing of NVMs across multiple nodes, as well as easy system hardware upgrades or re-configuration. Such a storage serves to complement the HPC center's PFS and provides intermediate staging area. Thus, the storage system is required to be as lightweight as possible and not be burdened by traditional PFS overhead. This is the reason we did not choose PVFS [4] or Lustre [6] for the aggregate NVM store.

In this paper, we use the aggregated NVM storage to provide memory extension for data-intensive computing on multi-core systems. The aggregate space can be presented using a separate partition of "fat nodes", equipped with NVM to provide uniform local accesses. Alternatively, the aggregate space can be built dynamically, on a per-job basis, using a subset of the job's allocated nodes that are equipped with NVM. We examine both these models in our proof-of-concept prototype development and its evaluation.

III. Design and Implementation

A. Goals

We are guided by the following goals in designing NVMalloc.

Providing explicit control to applications via familiar interfaces: Scientific applications should be able to explicitly manipulate the collective NVM storage just as they would operate on DRAM allocations. This implies the ability to control individual dataset's placement (on DRAM or NVM), allocate storage space from the aggregate NVM store and have it appear as memory buffers, and free the allocated buffers.

Transparent access to local and remote NVM alike: In our target environment, only a subset of the compute nodes might be equipped with NVM and nodes in this set may or may not overlap with a job's own node allocation (depending on whether we have an aggregate NVM storage that is center-wide or per-job.) From a compute client's standpoint, it should not be required to be aware of the location of NVM devices and be able to use any NVM-resident variables in the same way as DRAM variables.

Bridging byte-addressability and block storage: A key requirement for NVMalloc is to address the granularity mismatch between the byte-addressable `mmap` interface and the distributed block storage. Since memory accesses are byte-based, the `mmap` interface will send out I/O requests to the underlying storage for each and every access. Even when the NVM devices are local, the overhead can be significant, let alone remote accesses within the distributed NVM store. Meanwhile, block storage accesses perform I/O in larger granularity due to the latency involved in accessing block devices.

Optimizing NVM performance and lifetime: Byte-addressability is one aspect towards overall performance. Since we intend to provide the collective NVM storage to expand memory for applications that will use it in conjunction with DRAM, it is essential to optimize its performance through smart data placement. Even the fastest commodity NVM device today is more than an order of magnitude slower than DRAM. Therefore, we need to optimize the NVM store by taking into account the locality of the NVM, data access patterns, etc. Meanwhile, NVM devices such

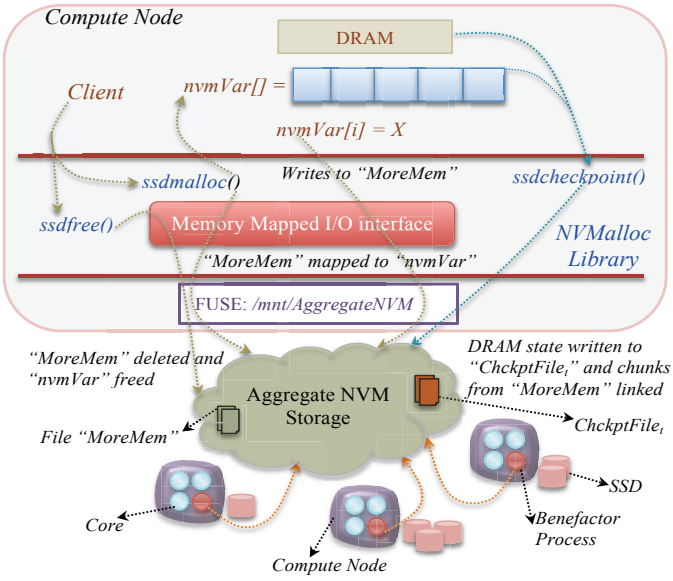

Figure 1. NVMalloc overview

as SSDs have limited write cycles. Our design needs to optimize the total write volume on these devices.

Ability to seamlessly checkpoint the memory-mapped variable: HPC applications routinely checkpoint their computation states. For a seamless user experience, we need to reconcile the checkpointing of an application's DRAM-allocated variables (or state) along with variables residing on the NVM device, while taking advantage of the persistent storage offered by NVM.

B. Architecture Overview

In order to accomplish the desired goals, we have focused our efforts on two fronts. First is a middleware layer, `NVMalloc`, comprising of a suite of services that enable clients to perform memory operations on the aggregated NVM storage system. Second is a set of modifications to the distributed NVM storage to make it amenable to be used as a memory partition. Figure 1 illustrates this usage model, wherein a parallel application's processes, potentially running on thousands of compute nodes, explicitly utilize the aggregate NVM store as an extended memory partition.

The primary motivation of NVMalloc is to let applications explicitly control the usage of the NVM store through the use of memory-mapped variables. To this end, out-of-core client applications can use NVMalloc to allocate memory from the NVM store for certain variables. These might include variables that are write-once-read-many or accessed infrequently, relative to the ones allocated on DRAM, which offers better access performance due to lower latencies and higher bandwidth. Alternatively, an application may not distinguish its variables in this fashion and might simply need more memory than physically available.

Our architecture comprises of several layers as shown in Figure 1. At the highest level is the application, requesting memory allocations from the NVM. Beneath that is the NVMalloc library, which enables explicit manipulation of the underlying NVM space. At the next level is the FUSE layer that provides a file system interface for NVMalloc so that the aggregated space can be accessed seamlessly. All of the above client-side components will reside on every compute node of a cluster. The lowest layer is the aggregate NVM store that abstracts the distributed NVM devices. This server-side component sits on compute or a subset of "fat" nodes with node-local NVM to contribute to the unified NVM space through a *benefactor* process.

C. Memory Mapping Files on Distributed NVM Storage

The `NVMalloc` layer supports a set of services such as `ssdmalloc()` and `ssdfree()` that lets a client allocate and free a block of memory from the aggregate NVM storage. Thematic to our design is the use of the POSIX `mmap()` interface that allows files or devices to be mapped into memory. Under the covers, the `ssdmalloc()` interface uses the `mmap()` system call, which maps a file, residing on the distributed NVM storage, onto an address space of the client process (Figure 1).

However, before we can memory-map a file on the distributed NVM, we need to be able to access the NVM store in a seamless file system-like fashion. To this end, we exploit our prior work on FUSE-enabling the distributed, aggregate NVM storage [11], [2] and extend it by implementing several file system flags necessary for interfacing with `mmap()`. Each compute node in our target environment has the aggregate NVM storage mounted via the FUSE user space file system (e.g., `/mnt/AggregateNVM`), which allows clients to open, read and write logical files that are stored in a distributed fashion. The file itself is striped across distributed NVM devices as chunks (256KB). Clients can invoke standard POSIX file system calls to operate on the file in the distributed NVM storage space.

One example of the flags implemented within our FUSE file system is O_RDWR, which opens a file in read/write mode and ensures that data written to it is available immediately for reading. This is needed by NVMalloc to make sure that the modified memory regions are available for immediate read accesses.

The pseudocode sequence below gives a simplified view of the mechanism behind an `ssdmalloc()` call:

```
fd = open("/mnt/AggregateNVM/MoreMem"...)
nvmVar = mmap(0, len, prot, flags, fd,
offset)
```

This maps a file, `MoreMem`, pointed to by `fd`, onto the client application's address space, beginning at `nvmVar`. Thereafter, addresses within the range, `[nvmVar, nvmVar+len-1]`, are legitimate addresses for the client application, mapped to the range of bytes `[offset, offset+len-1]` in the file on the NVM

store. Thus, `ssdmalloc()` returns a virtual memory address space that is mapped to an SSD-resident file (either on local or distributed devices.) Each ssdmalloc call creates a file on the NVM store, with an automatically generated file name internal to NVMalloc. The client need not be aware of the file name and only sees the memory-mapped variable, $nvmVar$. The prot argument needs to be set to $PROT\_WRITE|PROT\_READ$ to indicate the combination of read/write accesses on the data being mapped, which translates to a file that is also readable as well as writable at the backend NVM storage. The flags argument indicates the disposition of write references to the memory object. This needs to be set to MAP_SHARED to indicate that write references shall change the underlying file object. The alternative, $MAP\_PRIVATE$ flag, creates a copy on write and does not affect the original file. In particular, if the NVM-allocated variable, $nvmVar$, is to be checkpointed, the MAP_SHARED mode is essential. Reads and writes to the address range `[nvmVar, nvmVar+len-1]` are transformed by `mmap` into reads and writes to the file that is striped on the distributed NVM store.

In response to an initial `ssdmalloc()` request, the aggregate NVM storage performs file creation as follows. First, the manager on the aggregate NVM store generates a benefactor list (selected nodes equipped with NVM) that will store the file chunks and performs NVM space allocation on them (e.g., deducts their NVM space contributions to accommodate the size of the in-coming file.) The file creation is simply a space reservation and does not involve any data transfer immediately. The ssdmalloc buffer size is intimated to the distributed storage using `posix_fallocate()`, which tells the aggregate NVM manager to create appropriate file size metadata. Typically, `posix_fallocate()` is part of an advisory information option used to ensure that writes to the named file does not fail due to the lack of free space on the storage media. Subsequently, the benefactors are ready to receive data from the client in parallel. Actual data transfers occur between the client and the benefactors when `mmap()` issues read and write calls. From then on, a client application can operate on the NVM storage using the virtual address `nvmVar`.

The `ssdfree()` call uses the `munmap()` system call to release mappings to the file. Further, the memory-mapped file on the aggregate NVM will be deleted. If it has not been explicitly checkpointed (discussed in Section III-E) NVMalloc or the aggregate store offers no guarantees that the file will be persistent. One can imagine associating a *lifetime* with these memory-mapped variables, residing on the NVM store, so that they are persistent beyond the application run. Such a scheme can aid data sharing between a workflow of jobs or a simulation and its in-situ analysis.

D. Bridging the Granularity Gap

The `mmap` interface allows us to access a file residing on a block NVM store in a byte-addressable fashion. However, there is an obvious granularity gap between the byte-by-byte memory accesses and larger block accesses. Moreover, our distributed NVM storage system also delivers data in the form of larger chunks to clients to minimize the number of network requests. Thus, there is the need to bridge this gap to optimize performance.

To address this, we exploit the FUSE layer, beneath the NVMalloc library, on each compute node that is needed to access the aggregate NVM store. We use the FUSE cache to optimize both the read and write performance to the NVM-allocated variable. The FUSE client's cache size is a tunable parameter that can be adjusted at the time of instantiation. The cache size needs to be sufficient enough to aid with bridging the granularity gap, while also not consuming too much DRAM. The cache size used in our tests is 64 MB.

The NVM variable read operation in our library works as follows. When a read access is made (*e.g.*, with `x = nvmVar[i]`), `mmap` resolves the index into a corresponding POSIX read call for the particular offset into the file, `"/mnt/AggregateNVM/MoreMem"`, which is then propagated to the FUSE layer. The read implementation within the FUSE client transforms the operation into calls understood by the aggregate NVM store. With NVMalloc, benefactors store chunks as individual files and an NVM variable can be spread across multiple chunk files. Therefore, a call is first issued to the NVM manager to decipher which benefactor stores the requested data chunk. Next, the FUSE client makes a direct connection to the appropriate benefactor to retrieve the data chunk needed. The default chunk size used for this communication is 256 KB. Thus, for each byte of access, a 256 KB chunk will be fetched from a remote benefactor, which can be quite expensive if we do not perform caching at the FUSE layer. Caching the chunks at the FUSE layer can significantly aid in data reuse and minimize network data transfers. We exploit this trait of the FUSE buffers to improve the overall read performance within the NVMalloc library. However, it should be noted that this approach primarily helps the sequential read access pattern, which is the predominant usecase in HPC settings.

The NVM variable write works as follows. Upon a write request, the call is resolved, as in the read case, to the FUSE layer and the corresponding chunk to be updated is read from the benefactor to the FUSE client's cache in case of a miss. The 256KB chunk includes 64 pages (4KB) and the OS page cache sends out write requests to the FUSE layer on a page granularity. After this, we mark the page as dirty within the FUSE cache. The 64MB FUSE cache is managed using LRU. Upon chunk eviction, NVMalloc performs optimized writing, by sending only the dirty pages from within that chunk to appropriate target benefactors,

avoiding unnecessary transfers of clean pages within the chunk. This process is repeated until enough dirty pages are evicted to make room for the incoming chunk. If there are not enough dirty pages to evict to accommodate the new chunk, then the oldest chunk is evicted. Thus, for writes, the FUSE cache is optimized on a page-level within the chunks.

E. Seamless Checkpointing of DRAM and NVM Variables

Checkpointing is an I/O operation that saves the current state of an application's execution, mainly to protect it from failure. Checkpoint files are used to restart an application from a previous consistent state. We are interested in studying the interplay between application checkpointing and NVMalloc. When it comes to saving the state at every checkpoint timestep, an application needs to snapshot both DRAM and NVM allocated variables alike.

The process of checkpointing DRAM will typically produce a restart file on the PFS. However, in our setting, given the availability of an aggregate NVM store, it is only fair to assume that the application will want to checkpoint to this fast storage instead of the traditional disk-based PFS. In fact, we have previously shown that checkpointing to such an intermediate device and draining to PFS in the background is an extremely viable alternative and can help alleviate the I/O bottleneck [11], [21]. An application can easily open a checkpoint file on the distributed NVM store (e.g., `/mnt/AggregateNVM/CheckpointFile`) and write to it using standard POSIX I/O through our FUSE-based file system. In such a scenario, the checkpoint file is also distributed (striped) across many nodes, much like the memory-mapped variable. The question then is how to reconcile these two, to provide a logical restart file for the client, and to optimize the checkpointing performance/volume in the context of NVMalloc.

To this end, NVMalloc library provides the `ssdcheckpoint()` service. An application can use `ssdcheckpoint()` to checkpoint both its DRAM variables (and execution state) as well as NVM variables into one logical restart file. The `ssdcheckpoint()` service is a transparent interface that essentially dumps the entire application state to the aggregate NVM store as a file. This method copies the entire physical memory address space, followed by the NVM-allocated memory-mapped variables. This process will begin to create the chunks for the checkpoint file on the NVM store. Let the checkpoint file at timestep, t, be, $chckptFile_t$. Let the chunks created to checkpoint DRAM-resident data be $\{a, b, c\}$. These chunks now reside on the NVM benefactors. The memory-mapped variable is already persistently stored on the aggregate storage and, therefore we should try to avoid redundant copying. Let the memory-mapped variable, `nvmVar`, point to the file, $FileForNvmVar$, which in turn points to the chunks $\{d, e, f\}$. The scope of nvmVar is within the client application. The file, $FileForNvmVar$, and its chunks reside on the aggregate store. To avoid redundant copies on the aggregate storage, we link together the chunks of the `nvmVar` at the end of $chckptFile_t$. Appropriate chunk-mapping metadata is updated in the manager for $chckptFile_t$ to reflect this merge. After this process, $chckptFile_t$ points to chunks $\{a, b, c, d, e, f\}$. Here, we have shown a basic layout for the checkpoint file. However, a user may wish to specify the layout of the variables within the checkpoint file. Such information can be potentially passed through the `ssdcheckpoint()` interface.

The above solution avoids unnecessary copying of the NVM allocated variables, saving both checkpointing cost and NVM write cycles. However, this creates a challenge for subsequent accesses to these variables. $nvmVar$ is stored as chunks on the aggregate NVM store, and $chckptFile_t$ essentially reused those chunks. We need to ensure that subsequent modifications to $nvmVar$, during the following compute phases, do not alter $chckptFile_t$.

We need a way to optimally store the modifications to `nvmVar`, during the compute phase, between any two checkpoints. One approach is to create a new backend file (as chunks on the aggregate NVM store) that is a copy of the original variable at the time of the checkpoint. However, this defeats our purpose of avoiding unnecessary duplicates and is inefficient. Alternatively, we adopt a *copy-on-write scheme for data access and checkpointing*.

With our approach, write operations to `nvmVar` will be resolved to a chunk. For example, if chunk e is modified out of `nvmVar`'s $\{d, e, f\}$ chunks, we create a new chunk, e' (with the modifications), on the aggregate store. Now the memory-mapped file, $FileForNvmVar$ for `nvmVar`, contains the chunk set $\{d, e', f\}$. Note that $chckptFile_t$ and $FileForNvmVar$ still share the other unmodified chunks, $\{d, f\}$. In case of failure, the application can be restarted from $chckptFile_t$. At the next checkpoint timestep, $t + 1$, a new checkpoint file, $chckptFile_{t+1}$, will be created with chunks corresponding to any physical memory data and `nvmVar`'s chunks, $\{d, e', f\}$. This way, we ensure that the checkpoint files and the NVM-allocated variables can share chunks whenever possible and yet retain the ability to modify the memory-mapped variables during the compute phases. What is more, incremental checkpointing is automatically enabled with the NVM store, further reducing write overhead and wearing.

IV. Evaluation

We have evaluated the NVMalloc library that we have built atop an aggregate NVM store.

A. Testbed

Our experiments were conducted on the 128-core HAL cluster at Oak Ridge National Lab. Detailed configuration of the cluster is shown in Table II. Each compute node runs

Table II
TESTBED: HAL CLUSTER.

Type	HAL cluster
Compute nodes (#)	16
Cores per node (#)	8
Processor (GHz)	2.4
Memory per node	8GB
SATA SSD model	Intel X-25E, 32GB
Network	Bonded Dual Gigabit Ethernet

the Linux 2.6.32 kernel and is equipped with SATA Intel X-25E SSD. The specification of the SSD is presented in Table I. The aggregate NVM store is built by running a benefactor process on a core/node, on a subset of the nodes.

B. Performance Analysis

We analyzed NVMalloc as follows. We measured an application run that allocates certain variables through NVMalloc and compared it against a run when all the variables are allocated on local DRAM. To be fair, we show results from different application access patterns to study when it does and does not make sense to use NVMalloc.

1) STREAM: STREAM [13] is a widely used synthetic benchmark that measures the sustained memory bandwidth and computation rate for simple vector kernels, namely COPY, SCALE, SUM and TRIAD. In this paper, we present the TRIAD kernel results, whose computation kernel is given below. The other kernels produce similar results.

```
for(t = 0; t < TIMES; t++) {
    for(i = 0; i < N; i++) // TRIAD
        A[i] = B[i]+3*C[i];
}
```

We evaluated NVMalloc atop the distributed SSDs with STREAM and experimented with a variety of data placement settings (i.e., different combinations of arrays on NVM store and DRAM.) As is evident, the STREAM kernel tests the basic streaming of an array from/to the NVM store and does not perform any significant computation. Further, there is no reuse of data. Thus, this benchmark is intended to test the worst case performance of NVMalloc.

The STREAM kernel (with 8 threads) ran on a single node (with 8 cores.) The size of each array is 2GB and the kernel iterates 10 times. Figure 2 shows the TRIAD results, comparing the DRAM only mode and the NVMalloc mode on local and remote SSDs. Both the local and remote SSD accesses were handled transparently through the FUSE-based distributed NVM store. For the NVMalloc results, we tested 6 different TRIAD array placement options, by allocating a subset of arrays (such as "A" only or "B&C" together) on SSDs. Overall, we see that NVMalloc-based STREAM performance (both local and remote SSDs) significantly falls behind the DRAM performance by a factor of 62 and 115 for local and remote SSDs, respectively.

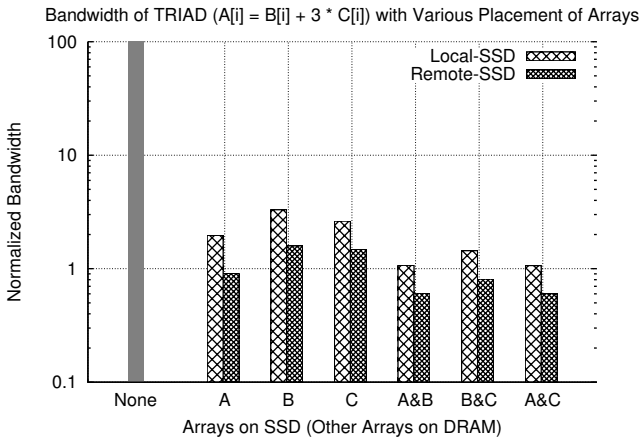

Figure 2. Bandwidth of STREAM TRIAD. Y-axis is normalized to show the bandwidth measured in the DRAM-only mode as 100. Note that a logarithmic scale is used for the Y-axis. X-axis shows which arrays are on the NVM. "None" indicates all arrays are on DRAM.

Table III
TABLE SHOWS THE BANDWIDTH (MB/S) OF STREAM WITH ARRAY C ON LOCAL SSD, WITH AND WITHOUT NVMALLOC. OTHER DATA PLACEMENT OPTIONS SHOW SIMILAR BEHAVIOR.

STREAM Kernel	COPY	SCALE	COPY	TRIAD
w/ NVMalloc	176	237	263	340
w/o NVMalloc	117	187	170	289

The results are not surprising, however, given that STREAM benchmarks the raw device bandwidth and there is at least a factor of 40 bandwidth difference between DRAM and the SSD models we tested. What this experiment suggests is that if all an application does is to stream data through NVMalloc and not perform any intelligent computation, it is obviously going to suffer a significant performance hit. Meanwhile, further experiments suggest that the NVMalloc framework does not introduce significant overhead itself. In fact, STREAM accesses to local SSD with and without NVMalloc (Table III) suggest that NVMalloc actually improves performance by introducing another layer of FUSE-based read-ahead caching that we have implemented. Since the STREAM access is sequential, the larger chunks fetched by NVMalloc into the FUSE layer helps subsequent accesses.

2) Matrix Multiplication: For the next set of experiments, we used an MPI implementation of dense matrix multiplication (MM), a resource-intensive kernel, which possesses computation and memory access patterns common in numerical simulations. MM computes $C = A \times B$, where A, B and C are $n \times n$ matrices. Both A and B are stored contiguously in an input file. Like in many applications, only one master process reads the input file and broadcasts the data to all the

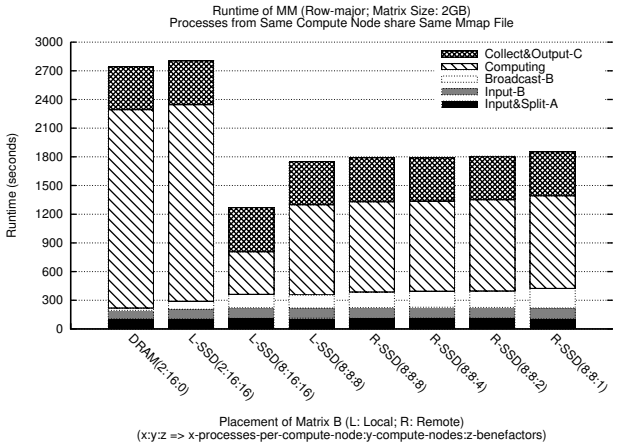

Figure 3. MM runtime with shared `mmap` file for matrix B for the problem size of 2GB/matrix.

other processes for parallel multiplication, using a BLOCK distribution. Input and output files, one for each matrix, are stored in a PFS.

MM's execution is broken down into five steps (shown in Figure 3 and 4): (i) the master MPI process reads A from an input file and sends chunks (partitioned in row or column order) to the slave processes; (ii) the master reads B; (iii) the master broadcasts B to all processes; (iv) all processes compute their local C partitions; (v) the master gathers and writes the resulting C partitions. Thus, A and C are distributed among all the processes, while B is fully replicated. We implemented MM with loop tiling [30], a common optimization that partitions the main loops' iteration space for better cache reuse.

In our experiments, we varied the following: (i) tiling size, (ii) data access patterns (row-major or column-major access for the SSD-resident matrix), and (iii) the matrix placement (which matrix or matrices to place on NVM.) For a fair comparison, we disabled all swapping files or devices, and locked enough memory through $mlock()$ to leave only $1.25GB$ memory for the system (including space for the kernel or underlying file system cache/buffer). We also tested with two other options ($1.75GB$ and $2.25GB$) but did not see significant difference in our experiments and report numbers with $1.25GB$ memory for the system.

As all processes share the same matrix B, which remains read-only after being initialized, an obvious optimization here is to allow multiple processes within a compute node to map their matrix B to a shared file, residing on the NVM store. This option saves both storage space, I/O and network traffic and is enabled by a special flag to the ssdmalloc() interface within the NVMalloc library.

Figure 3 shows the results for a size of 2GB for each of the matrices, accessed in a row-major fashion. The figure also depicts multiple memory allocation and storage distribution settings, with the total execution time of each job broken down into the five aforementioned stages. The MM algorithm we tested has excellent computation scalability. Since matrix B is replicated across all processes, each process can proceed with its computation, requiring little communication with its peers. On the other hand, this approach has higher memory consumption (compared to alternatives such as decomposing both A and B.) With the amount of physical memory available (8 GB/node), the DRAM-only solution could only fit two processes on each node, wasting 75% of the compute power on these 8-core machines. With NVMalloc, SSDs can be used to seamlessly extend the DRAM space, allowing all of the cores to be utilized.

For a direct comparison, we carried out experiments where only 2 processes are allocated on each compute node. This result is shown by the L-SSD(2:16:16) bar in Figure 3 (where "L" stands for "local", and "2:16:16" indicates that there are 2 processes per node, 16 nodes used in the job, and 16 SSD benefactors). In this case, the overall performance with NVMalloc is only slightly worse (by 2.19%) than using DRAM only, due to an increase in the cost of broadcasting B. The computation, which consumes the bulk of the total execution, appears to take the same amount of time when the majority of data structures are allocated on the SSD. This suggests that commodity SSD units can be effectively used as a memory extension in high-performance computing applications, with their higher access latency hidden automatically by multiple layers of memory caches in the system. The result further indicates that we can even run larger problem sizes than what the DRAM can accommodate.

Next, we examined the performance of using 8 processes per node to maximize core utilization (Figure 3). NVMalloc enables each process to occupy a smaller footprint on DRAM. L-SSD(8:16:16) achieves a 53.75% improvement compared to the DRAM only case. Again, the computation stage shows excellent scalability, relative to the total number of processes. We then analyzed the use of remote SSDs from the NVM store. For these tests, the SSD benefactors were all remote to the compute nodes. We can see that there is very little overhead (1.42%) in using remote SSDs as shown by cases L-SSD(8:8:8) and R-SSD(8:8:8). Further, R-SSD(8:8:8) is still 34.73% better than the DRAM only case, suggesting that network does not appear to be a bottleneck.

The group of results using R-SSD(8:8:z) settings in Figure 3 tested the performance of MM under varied ratios of compute node to SSD benefactor nodes in the NVM store. This can shed light on the number of clients an SSD benefactor can serve. We can see that reducing the number of SSD units equipped on the cluster (with each shared by more compute nodes) does not have a visible effect on most stages of the MM execution, except for slight increases again in the broadcasting stage due to higher I/O and network traffic concentration. In particular, the R-SSD(8:8:1) results reveal that by adding *one* $300 SSD drive to every 8 compute

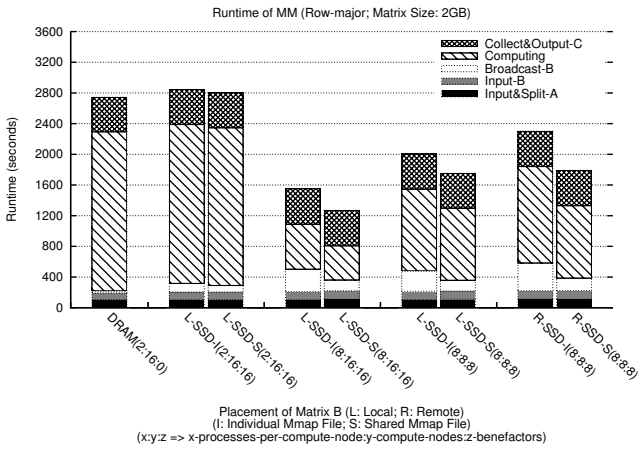

Figure 4. MM: shared versus individual `mmap` files for matrix B for a problem size of 2GB/matrix.

nodes and using mechanisms like NVMalloc, we can bring about a 32.47% performance improvement while running on *half* the nodes compared to the DRAM only mode. This result suggests that future machines can reduce the total provisioning cost by purchasing a combination of DRAM and NVM and use them in concert as above.

Shared versus Individual `mmap` files: To observe the performance when processes do not share common read-only data structures, we also evaluated the case where each process maps its matrix B to a separate file. In Figure 4, bars labeled "-SSD-S" use shared `mmap` access, while "-SSD-I" uses per-process `mmap` files. Not surprisingly, the individual mode is slower (up to 18%). Key factors include the increased broadcasting and computation overhead. The performance difference is particularly more, when all 8 cores are used (the "(8:y:z)" cases), with larger memory and I/O contention. However, the individual file mode still has significant advantages over the DRAM case.

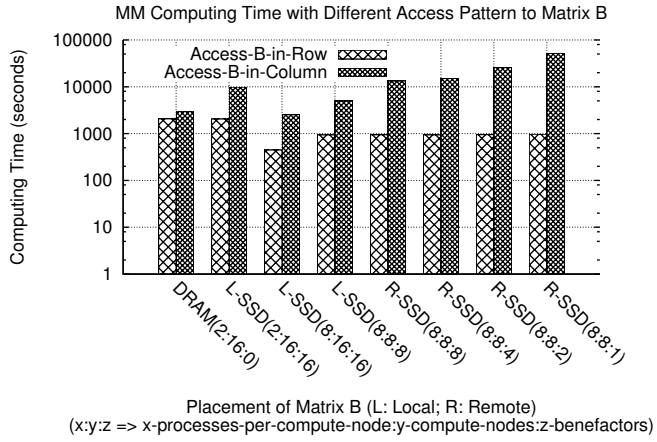

Figure 5. MM: row-major versus column-major for 2GB/matrix.

Table IV
DATA EXCHANGED BETWEEN APPLICATION, FUSE AND SSD STORE FOR A PROBLEM SIZE OF 2GB/MATRIX.

Access Pattern of B	Aggregated Accesses to B (GB)	Request to FUSE (GB)	Request to SSD (GB)
Row-major	256	4	2
Column-major	256	113	130

Table V
PERFORMANCE (COMPUTING TIME IN SECONDS) OF MM (L-SSD(8:16:16)) WITH VARIOUS TILE SIZE FOR 2GB/MATRIX.

Tile Size	16×16	32×32	64×64	128×128
Row-major	464	449	446	443
Column-major	4190	2628	2549	1325

Row and Column-major Accesses: To evaluate the impact of memory access pattern, we experimented with two access orders for B: column-major and row-major (effectively altering the data placement strategy). Figure 5 shows the runtime of the computation phase. As expected, column-major is much slower. Further, its performance degrades significantly when the SSD resources are reduced (from L to R, then with declining number of benefactors), while the row major performance remains stable. Also, the difference between row- and column-major performance is much more pronounced with NVMalloc-enabled memory extension compared to the DRAM only case. The explosion in column major execution time is due to a combination of factors, including less data locality for DRAM caching, random SSD accesses, and more network communication as well as I/O volumes. For row-major accesses, our FUSE-based read caching helps due to the locality of accesses. Essentially, a sub-optimal access pattern can dramatically weaken the capability of hiding SSD access latencies with DRAM caches, exposing the inferior capability of SSD-based memory extension. This indicates that applications need to be aware of the ramifications of the data access patterns and the placement of their data structures while using the NVMalloc library.

Table IV shows the size of the aggregated data read during the computing phase of MM, the size of the data requested from the system to the FUSE buffers, and the real transfers between the compute node and the SSD-benefactor-node, for both row-major and column-major L-SSD(8:16:16) cases. From the table, the SSD access latency can be effectively hidden by the caching mechanism in NVMalloc if there exists good access locality (row-major). This is also shown by the performance comparison between row-major and column-major results (Figure 5).

Varying the tiling size: We also varied the titling size of

Table VI
SORTING TIME (IN SECONDS) WITH VARIOUS CONFIGURATION.

Quicksort	DRAM (8:16:0)	L-SSD (8:16:16)	R-SSD (8:8:8)
Time (s)	1148.82	100.57	301.24
Pass (#)	2	1	1

MM to further study memory access patterns. The matrix size is 2GB, which means each process has 128 rows or columns to calculate. Thus, the largest tile size possible is 128×128. For column-major accesses, Table V shows that as the tile size increases, the computing time decreases, indicating locality of accesses within larger tiles. For row-major accesses, however, we did not see a significant improvement due to larger tiles, due to inherent sequentiality.

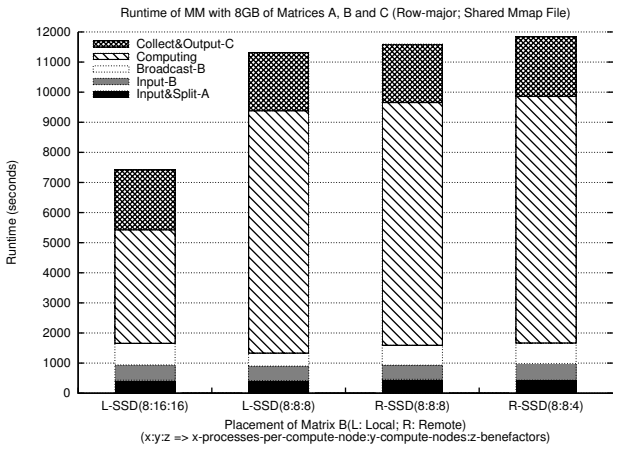

Figure 6. MM with 8GB/matrix problem size.

8GB Problem Size: To illustrate the potential to run applications with problem sizes larger than what the physical memory allows, we increased the size of the matrices to 8GB each (Figure 6.) Note that the physical memory size is only 8GB/node. Matrix B is accessed from the NVM store using a shared `mmap` file, while matrices A and C are in DRAM, split between the processes in each configuration. Comparing against the 2GB computation phase, the 8GB computation should have increased by a factor of 16. However, the loop tiling technique favors computing with longer rows (the 8GB case) much more than shorter rows and the computing phase only increases by a factor of 9. Thus, the performance due to NVMalloc scales well for larger sizes.

3) Parallel Sorting: In this experiment, we used an MPI based parallel *Quicksort* program. Quicksort sorts a list of data elements based on the divide and conquer strategy, splitting the entire list into two sub-lists and sorting them recursively. The basic steps of the algorithm are as follows: (i) Choose a pivot element in the list. (ii) Reorder the list such that elements less than the pivot are arranged before the pivot and all elements greater than the pivot come after it. After the partitioning, the pivot is in its final position. (iii) Recursively reorder two sub-lists.

Table VII
DATA EXCHANGED DUE TO NVMALLOC WRITE OPTIMIZATION FOR A RANDOM WRITE SYNTHETIC APPLICATION.

NVMalloc write optimization	Data Written to FUSE	Data Written to SSD
w/ Optimization	467MB	504MB
w/o Optimization	471MB	19.3GB

Table VI presents the results of the MPI based Quicksort program. In these experiments, we measured the total running time of a 200 GB data list for various DRAM/NVM configurations. DRAM(8:16:0) is a configuration that runs on the entire machine (8 cores/node and 16 nodes), using all of the system memory (128 GB). Even so, it does not have sufficient memory to load all of the data (200GB) at once. L-SSD(8:16:16) is a hybrid DRAM + SSD store configuration, where 100 GB of data are loaded on the DRAM, and the other 100 GB are loaded on 16 local SSDs, using NVMalloc. R-SSD(8:8:8) is also a hybrid DRAM + SSD configuration, where 50 GB of data are loaded on the DRAM, and the other 150 GB are loaded on 8 remote SSDs, using NVMalloc. The results show that L-SSD(8:16:16) offers the best performance for this setting, providing a factor of 10 speedup compared to DRAM(8:16:0). Since DRAM(8:16:0) is unable to load the entire 200 GB dataset, it requires us to change the original program to decompose the entire dataset into two sub datasets and run two passes to sort the sub datasets. The two passes of DRAM(8:16:0) also require significant data exchange between each other, with the PFS used to share the interim sorted data. These steps are obviously not required for both L-SSD(8:16:16) and R-SSD(8:8:8). R-SSD(8:8:8) is slower than L-SSD(8:16:16), since it has half the number of nodes with double the workload. This experiment illustrates that NVMalloc is able to run problems much larger than what the physical memory allows.

4) Write Optimization in NVMalloc: With MM, we showed the benefits of read caching within NVMalloc. To study the write optimization, we constructed a synthetic application that issues write operations (128K times) to randomly generated address within a 2GB data located on the SSD store. Writes were issued byte-by-byte to these random addresses, to depict the performance of our optimization under an extreme case. Table VII shows the results with and without our optimization. For each dirty chunk, rather than sending the entire chunk (256KB) to the local or remote SSD, writing only the dirty pages (4KB) significantly reduces the data transferred between FUSE and the SSD.

5) Checkpointing DRAM/NVM Variables: We study the benefits of seamless checkpointing of DRAM/NVM vari-

Table VIII
PERFORMANCE OF SEAMLESS CHECKPOINTING

DRAM/NVM Variables checkpointing	Test Case 1	Test Case 2	Test Case 3	Test Case 4
DRAM size (GB)	8	8	8	8
NVM size (GB)	4	8	16	32
Regular chkpt time (s)	248.6	437.3	866.4	1633.5
SSDcheckpoint time (s)	82.4	82.4	82.4	82.4
Improvement in time	66.8%	81.2%	90.5%	95.0%
Improvement in size	33.3%	50.0%	66.7%	80.0%

ables. We varied the size of NVM variables from 4GB to 32GB and fixed the size of DRAM variables at 8GB. We used 112 clients, checkpointing in parallel, to the NVM store. We compared ssdcheckpoint that avoids the redundant copy using incremental writes against regular checkpointing that re-writes the NVM-allocated variable to the checkpoint file. For both `ssdcheckpoint` and regular checkpointing, the checkpoint file is saved in the aggregate NVM store.

Table VIII provides the average performance improvement of ssdcheckpoint over regular checkpointing. The checkpointing cost mainly contains two parts: (1) saving the DRAM state to the checkpoint file and (2) saving the memory-mapped file on NVM to the same checkpoint file. The time taken to save the DRAM state is obviously the same for both ssdcheckpoint and regular checkpointing. However, for the NVM variables, our ssdcheckpoint mechanism enables the sharing of chunks between the checkpoint file and the memory-mapped files of the NVM-allocated variables. Therefore, the entire overhead of duplicating data is avoided. Thus, as Table VIII shows, the larger the NVM memory to checkpoint, the better the improvement. Note that the improvement is more significant than the portion of memory allocated on the NVM store. This is because checkpointing NVM-resident variables with the regular approach is more costly, as NVM reads are slower than DRAM reads.

V. RELATED WORK

Paging is a popular memory management technique in modern operating systems that enables memory pages to be swapped in and out between DRAM and I/O devices [24]. HDD can have virtual memory partitions for swap spaces. Recent popularity of NAND flash memory has enabled the exploration of swapping on SSDs. To this end, several solutions have been proposed [10], [20], [12], [23], [25]. Ko et. al. [10] have proposed a log-structured swapping algorithm to avoid performance degradation due to garbage collection. Park et. al. [20] have proposed a flash aware page replacement algorithm that can avoid the high performance penalty of erase operations on NAND flash. Also there have been several empirical studies on how to use a flash device, connected over the network (such as Infiniband) or the PCIe bus, as a swap area [12], [25]. All of these explore the following unique properties of NAND flash unlike magnetic disks: (i) writes are expensive than reads, (ii) data access granularity is a page (2KB or 4KB), thus misaligned pages need to be carefully managed, and (iii) flash device has a life time concern on flash cells.

Large-scale HPC machines have swap space turned off as they do not contain node-local disks. Also, swapping to disk can potentially cause more unpredictable performance due to the memory to disk gap. Faster flash devices and the above optimization efforts can encourage the use of swap on flash in HPC. While NVMalloc shares the goal of providing more memory through flash, it is different in its attempt to provide explicit control over the extended partition. We provide applications more control on data placement. Further, all of the aforementioned efforts require changes to the operating systems, however, our approach resides at the user-level.

Accessing remote memory to supplement node-local DRAM has been explored previously [31]. This effort attempts to access additional memory from specialized memory servers through MPI communication panels. NVMalloc is similar in the sense that it `mmap`s from a distribute pool of devices (albeit NVM), but needs to deal with a variety of byte-addressability issues that network memory systems do not address.

In the recent work on ssdalloc [3], the authors have considered various application level approaches to use the flash device as an extended memory space on a single server-based system, and concluded using `mmap` on flash device can incur significant runtime overhead. In contrast, we aim to provide more memory for data-intensive parallel applications running on many-core systems. In particular, we address the unique challenge of building an aggregate store upon distributed SSD devices attached to a subset of supercomputer nodes. Our work also draws different conclusions on the utility of `mmap` based on our target domain and application access patterns.

Efforts such as Mnemosyne [28] and NV-heaps [7] strive to provide a persistent interface to second-generation NVM such as phase change memory (PCM), STT-RAM, memristors so that in-memory data structures such as trees, lists and hashes can survive system crashes. Our work on NVMalloc complements these efforts and while we have demonstrated it using the first-generation NVM (SSD), it is applicable to next-generation devices as well. Further, the byte-addressable, second generation Phase Change Memory (PCM), which is still a few years from mass production, cannot yet scale to the capacity levels of block-based NVM (e.g., PCM on DIMM has only been prototyped for up to 1GB; PCM on PCIe can scale further, but is slower.) Moreover, the power consumption for a baseline PCM is much more than DRAM. Thus, deep memory tiers are most likely to be presented on a subset of "fat" nodes with some node-local byte-addressable NVM as and when

that technology matures. Thus, our work on accessing a distributed NVM store through NVMalloc is very timely.

VI. Conclusion and Future Work

In this paper, we have presented the rationale, design and implementation of NVMalloc, a runtime library that supplies parallel applications with a distributed NVM storage as a secondary memory partition. NVMalloc provides a suite of services for applications to explicitly allocate and manipulate memory regions on the NVM store, bridge the gap between byte-addressable memory accesses and the block-based store, seamlessly checkpoint DRAM/NVM-resident variables and optimize for SSD-specific data access patterns. Our evaluation on a multicore testbed suggests that NVMalloc is viable. It further enables cost-effective parallel computation by allowing applications to (1) utilize the multiple cores available more efficiently in data-intensive computation, and (2) compute problem sizes much larger than what the physical memory permits.

In our future work, we plan to evaluate NVMalloc with more out-of-core applications, address proactive prefetching based on application memory allocation/access patterns, and explore transparent interfaces for applications to allocate large objects across the DRAM and NVM space.

Acknowledgments

We would like to thank the reviewers for their feedback. This work was sponsored in part by ORNL, managed by UT-Battelle, LLC for the U.S. DOE (Contract No. DE-AC05-00OR22725), by the U.S. NSF Awards CCF-0937827, CCF-0937690, CCF-0937908, and CNS-0958311, and by the joint appointments of Sudharshan Vazhkudai and Xiaosong Ma between ORNL/UT and ORNL/NCSU, respectively.

References

[1] Intel X25-E Extreme SATA Solid-State Drive. http://download.intel.com/design/flash/nand/extreme/extreme-sata-ssd-product-brief.pdf.

[2] S. Al-Kiswany, M. Ripeanu, S. S. Vazhkudai, and A. Gharaibeh. stdchk: A Checkpoint Storage System for Desktop Grid Computing. In *ICDCS'08*.

[3] V. S. P. Anirudh Badam. SSDAlloc: Hybrid SSD/RAM Memory Management Made Easy. In *NSDI'11*.

[4] P. Carns, W. L. III, R. Ross, and R. Thakur. PVFS: A Parallel File System For Linux Clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference*, 2000.

[5] F. Chen, D. A. Koufaty, and X. Zhang. Hystor: making the best use of solid state drives in high performance storage systems. In *ICS'11*.

[6] Cluster File Systems, Inc. Lustre: A scalable, high-performance file system. http://www.lustre.org/docs/-whitepaper.pdf, 2002.

[7] J. Coburn, A. Caufield, A. Akel, L. Grupp, R. Gupta, R. Jhala, and S. Swanson. NV-Heaps: Making Persistent Objects Fast and Safe with Next-generation, Non-volatile Memories. In *ASPLOS'11*.

[8] Fusion-io, Inc. ioDrive Duo. http://www.fusionio.com/-products/iodriveduo.

[9] Y. Kim, A. Gupta, B. Urgaonkar, P. Berman, and A. Sivasubramaniam. HybridStore: A Cost-Efficient, High-Performance Storage System Combining SSDs and HDDs. In *MASCOTS'11*.

[10] S. Ko, S. Jun, Y. Ryu, O. Kwon, and K. Koh. A New Linux Swap System for Flash Memory Storage Devices. In *ICCSA'08*.

[11] M. Li, S. S. Vazhkudai, A. R. Butt, F. Meng, X. Ma, Y. Kim, C. Engelmann, and G. M. Shipman. Functional Partitioning to Optimize End-to-End Performance on Many-core Architectures. In *SC'10*.

[12] S. Liang, R. Noronha, and D. K. Panda. Swapping to remote memory over InfiniBand: An Approach using a High Performance Network Block Device. In *Cluster'05*.

[13] J. D. McCalpin. Memory Bandwidth and Machine Balance in Current High Performance Computers. In *TCCA'95*.

[14] D. McGrath and M. LaPedus. 'Universal memory' race still on the starting block. http://www.eetimes.com/electronics-news/4080780/-Universal-memory-race-still-on-the-starting-block, 2008.

[15] J. C. Mogul, E. Argollo, M. Shah, and P. Faraboschi. Operating System Support for NVM+DRAM Hybrid Main Memory. In *HotOS'09*.

[16] D. Narayanan, E. Thereska, A. Donnelly, S. Elnikety, and A. Rowstron. Migrating Server Storage to SSDs: Analysis of Tradeoffs. In *EuroSys'09*.

[17] M. L. Norman and A. Snavely. Accelerating Data-intensive Science with Gordon and Dash. In *TG'10*.

[18] OCZ Technology Group, Inc. OCZ Revodrive. http://www.-ocztechnology.com/ocz-revodrive-pci-express-ssd.html.

[19] U. D. of Energy. DOE exascale initiative technical roadmap, December 2009. http://extremecomputing.labworks.org/hardware/collaboration/EI-RoadMapV21-SanDiego.pdf.

[20] S.-Y. Park, D. Jung, J.-U. Kang, J. Kim, and J. Lee. CFLRU: A Replacement Algorithm for Flash Memory. In *CASES'06*.

[21] R. Prabhakar, S. S. Vazhkudai, Y. Kim, A. R. Butt, M. Li, and M. Kandemir. Provisioning a Multi-tiered Data Staging Area for Extreme-Scale Machines. In *ICDCS'11*.

[22] D. Roberts, T. Kgil, and T. Mudge. Integrating NAND flash devices onto servers. *Communications of the ACM*, April 2009.

[23] M. Saxena and M. M. Swift. FlashVM: Virtual Memory Management on Flash. In *USENIX ATC'10*.

[24] A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating System Concepts*. Wiley, 2008.

[25] J. Suzuki, T. Baba, Y. Hidaka, J. Higuchi, N. Kami, S. Uchida, M. Takahashi, T. Sugawara, and T. Yoshikawa. Adaptive Memory System over Ethernet. In *HotStorage'10*.

[26] Top500 supercomputer sites. http://www.top500.org/.

[27] S. Vazhkudai, X. Ma, V. Freeh, J. Strickland, N. Tammineedi, and S. Scott. FreeLoader: Scavenging Desktop Storage Resources for Bulk, Transient Data. In *SC'05*.

[28] H. Volos, A. Tackcw, and M. Swift. Mnemosyne: Lightweight Persistent Memory. In *ASPLOS'11*.

[29] W. Wang, Z. Lin, W. Tang, W. Lee, S. Ethier, J. Lewandowski, G. Rewoldt, T. Hahm, and J. Manickam. Gyrokinetic Simulation of Global Turbulent Transport Properties in Tokamak Experiments. *Physics of Plasmas*, 13, 2006.

[30] M. Wolfe. More Iteration Space Tiling. In *SC'89*.

[31] C. Yue, R. T. Mills, A. Stathopoulos, and D. S. Nikolopoulos. Runtime Support for Memory Adaptation in Scientific Applications via Local Disk and Remote Memory. In *HPDC*, 2006.