The RAMDISK Storage Accelerator — A Method of Accelerating I/O Performance On HPC Systems Using RAMDISKs

Tim Wickberg, Christopher Carothers
Department of Computer Science
Rensselaer Polytechnic Institute
Troy, NY, USA
{wickbt,chrisc}@cs.rpi.edu

ABSTRACT

I/O performance in large-scale HPC systems has not kept pace with improvements in computational performance. This widening gap presents an opportunity to introduce a new layer into the HPC environment that specifically targets this divide. A *RAMDISK Storage Accelerator* (*RSA*) is proposed; a system leveraging the high-throughput and decreasing cost of DRAM to provide an application-transparent method for pre-staging input data and commit results back to a persistent disk storage system.

The RSA is constructed from a set of individual RSA nodes; each with large amounts of DRAM and a high-speed connection to the storage network. Memory from each node is made available through a dynamically constructed parallel filesystem to a compute job; data is asynchronously staged on to the RAMDISK ahead of compute job start and written back out to the persistent disk system after job completion. The RAMDISK provides very-high-speed, low-latency temporary storage that is dedicated to a specific job. Asynchronous data-staging frees the compute system from time that would otherwise be spent waiting for file I/O to finish at the start and end of execution. The *RSA Scheduler* is constructed to demonstrate this asynchronous data-staging model.

1. INTRODUCTION

Filesystem I/O performance in large-scale HPC systems has not kept pace with improvements in computational performance [11, 23]. Datasets continue to grow linearly with computational complexity and not with filesystem performance. This results in a decreasing amount of the time needed to run a compute job spent on the actual computation and an increasing amount spent waiting for I/O to finish [14]. As HPC systems continue towards Exascale there is growing concern that the current approaches to obtaining filesystem performance will not keep pace and that new architectures will be necessary [1, 4, 20].

We believe that an opportunity exists to introduce a new system layer that decouples the disk storage system from the compute system, one that is transparent to the compute application itself. The proposed solution is the introduction of a new component to the HPC systems architecture — the *RAMDISK Storage Accelerator* (*RSA*).

A RAMDISK can be constructed using a parallel filesystem by aggregating the DRAM available in a number of commodity servers. This parallel RAMDISK is exported to the compute system as if it were a traditional disk-backed filesystem. The RAMDISK provides for very-high-speed, low-latency access to temporary storage that is dedicated to a specific compute job. By asynchronously constructing the RAMDISK ahead of the start of the job execution, data is staged-in and ready to be quickly accessed at a much higher speed when the job begins. Similarly, by outputting data to the RAMDISK and waiting until after job execution finishes to asynchronously stage-out data to the disk storage system, the compute system is released and able to begin executing a separate job faster — freeing the system from from time that would otherwise be spent waiting for file I/O to finish.

The *RSA Scheduler* implements this asynchronous staging and manages the RSA nodes. It is tasked with anticipating when the next job will start, dynamically allocating RSA nodes to the job and provisioning a parallel RAMDISK filesystem on top of it, staging data in and out from the RAMDISK, and releasing resources after the data staging out step completes.

1.1 Related Work

A recent development using DRAM to provide a parallel filesystem entitled *RAMCloud* [18] first appeared in 2009. The RAMCloud implementation is designed to hold all filesystem data in memory and commits a copy of the data to dedicated disk storage on-demand.

Asynchronous data staging, in a similar manner to that implemented for the RSA, is also discussed in relation to different classes of disk storage as the *Zest* system [16]. In this model, data is written to an accelerated disk storage layer that directly pushes data back to a larger persistent storage layer and acts as an accelerated buffer, rather than the dedicated cache model of the RSA. Buffering mechanisms have also been discussed at the I/O node level on a Blue Gene/P system by using I/O node DRAM to provide temporary asynchronous file storage and allow the compute nodes in the machine to continue job execution while data is committed back to primary storage [23]. The *DataStager* system would provide a dynamic data staging platform using dedicated DataStager servers that share some similarities with the RSA nodes presented here [1]. However, DataStager relies on application modification to offload the I/O load, whereas the RSA presents a standard POSIX file interface. *IOFSL* [17] and *ZOID* [10] provide another approach to I/O acceleration by relying on internal modifications to the I/O nodes in large-scale compute systems to provide small-file I/O aggregation and a faster path back to the parallel filesystem.

The *Scalable Checkpoint/Restart* library provides another method of accelerating I/O on large-scale systems, especially Blue Gene systems [15]. By using storage available directly to each I/O node, in the form of SSDs, disk or RAMDISKs, it is able to provide higher-speed short-term caching of checkpoint data. The library is meant to operate independently of the primary storage, unlike the RSA and is meant for checkpoint data only, not end results. Additionally, by using the I/O nodes in the system, data still must be transferred back to persistent storage before the compute system can be released for the next job.

2. RAMDISK STORAGE ACCELERATOR ARCHITECTURE

The RAMDISK Storage Accelerator is constructed from a dedicated cluster of high-memory servers. *RSA nodes* are dynamically allocated to jobs prior to job execution on the compute system and a parallel RAMDISK is constructed. Jobs are able to *stage-in* data to the RAMDISK before the job begins execution on the compute system. Once execution begins the job can access data from the RAMDISK at a much higher speed than it would from the disk storage systems. Later on the job is able to make use of the RAMDISK again to write results data. Once data has been written to the RAMDISK the job can release its allocation on the compute system and the RSA will stage data out of the RAMDISK back to persistent storage.

A critical factor of the design is the asynchronous data-staging that the RSA nodes perform. To accomplish this, at any given time half of the RSA nodes should be expected to be serving active compute jobs, while the other half are either staging data in to a RAMDISK for a job that has yet to start or staging data back out for a job that has finished execution on the compute system.

Before a scheduled job is run on the compute cluster a segment of the RSA is allocated for that job in proportion to the requested compute system size and a RAMDISK is freshly created on the allocated nodes using a parallel filesystem. The parallel filesystem aggregates the DRAM available on each individual node into a single parallel RAMDISK. Data needed for the job is then staged-in to the RAMDISK. As this is happening independently from job execution, this transfer can occur at a much slower rate than would be required on the compute system itself.

Once the job starts execution on the compute system it reads its data in from the RSA at a much higher speed than it would from the disk storage. The RSA is then able to discard the data (as it remains in the persistent disk storage system) and reset to receive output from the compute system.

This reset occurs in the background without impacting the compute job. Once the transition has completed the now-empty RAMDISK can then be leveraged mid-execution for job snapshots and at the end of computation to write results out. After the job has written its results out to the RSA and completed execution, the compute system is then free to start the next job — it does not need wait for data to be pushed out to disk storage.

The RSA (still allocated to the finished job) then stages data back to the disk storage system. Again, the performance of the disk storage system no longer directly impacts the throughput of the compute system itself. An extended mode of operation permits the compute job to perform some data consolidation or post-processing independently of the main compute job. As an example, supposing the application wrote its results out to several thousand small files. These many small files could then be aggregated to a single file on the disk storage system instead. This addresses a recurring metadata performance issue in many HPC filesystems [2].

An additional advantage of this structure is that the I/O performance of the RSA scales linearly with compute job size — a task that is currently infeasible in traditional disk storage systems. Traditionally all compute jobs, outside file transfer processes, visualization systems and similar auxiliary services contend for access to the HPC center's filesystem, and complex interactions can severely reduce the overall performance [14]. Additionally, no major parallel filesystems currently provide quality-of-service methods that would allow administrators to control these interactions between systems contending for access. Generally, the only mediator is the relative network speeds of the various systems competing for access to the storage network.

Instead, the RSA nodes are directly allocated to the job which prevents contention for the throughput each RSA node can provide. Additionally, as RSA nodes are allocated in direct proportion to job size[1] there is a linear scaling between capacity and I/O performance of the RSA for each job, something that disk storage systems cannot currently provide.

2.1 RAMDISK filesystem

A parallel filesystem is used to construct the RAMDISK itself, allowing the aggregated memory of each RSA node to be made available to the I/O nodes in the compute system in a unified manner. Several options exist for this such as Ceph, Lustre, PVFS, or GPFS.

Parallel filesystem considerations and the specific implementation used for testing are available as part of the full Master's Thesis [25].

3. THE RSA SCHEDULER

For the RAMDISK Storage Accelerator to work efficiently certain stages in a job's lifecycle — job submission, job execution on the compute system, and cleanup — must be identified and managed separately. The four stages that are relevant for the RSA are: `on-deck`, `demoted`, `running` and `finished`. Transitions between these `job-state`s are shown in Figure 1.

[1]An extension to the RSA could provide additional RSA nodes to I/O intensive compute jobs on request, instead of relying on a directly proportion between compute nodes and the number of allocated RSA nodes.

The *RSA Scheduler* is constructed to coordinate these stages and manage the RSA nodes. It is implemented alongside the SLURM scheduler; the SLURM scheduler is tasked with managing access to the compute system and job status is then tracked through SLURM's APIs.

These transitions are discussed in depth in the Master's Thesis [25].

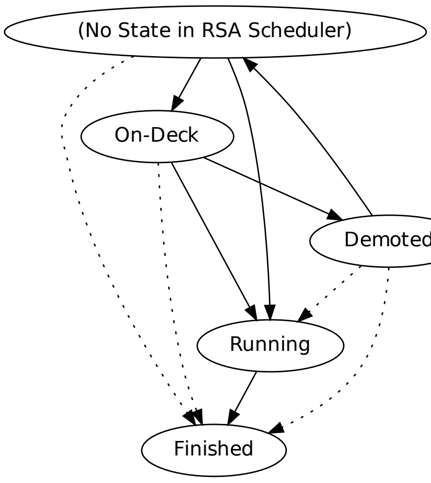

Figure 1: Job State changes, showing all possible `job-state` transitions. Dotted lines are unlikely state transitions.

3.1 RSA Setup and Data Staging In

One of the difficult steps faced by the RSA Scheduler is to determine which job is likely to start execution next on the compute system. In a production environment jobs will be continually added to the job queues, and job priorities will be constantly reassessed based on the scheduler's internal state, and jobs may be canceled or updated at any time by the system users. This dynamic scheduling environment presents a significant challenge on its own.

The implementation relies on SLURM to handle this task. The goal of the RSA Scheduler is not to re-implement a new production HPC job scheduler, but rather to add an additional level of capabilities to the compute system it manages. The scheduling information required by the RSA is limited to knowing when jobs start and finish and determining which jobs are likely to start execution next. This determination drives the initial RSA setup for a job and kicks off the data stage-in process. SLURM is relied on to make this determination; the RSA Scheduler learns the results by monitoring the `job-state` for each pending job through APIs to SLURM.

Pending jobs fall in two main categories in SLURM:

- Pending on job priority, internally denoted as a `job-state` of `PRIORITY`, where there are higher priority jobs waiting ahead of us.
- Pending on resources, `job-state` of `RESOURCES`, where the job is waiting for sufficient free compute resources to begin execution.

Note that jobs can fluidly move between these `job-state`s based on newly submitted jobs and other factors; the RSA scheduling must react in the event of these changes and reallocate the RSA resources to match these revised scheduling decisions. Thus, the implementation distinguishes between jobs that are `on-deck` — those next in line to begin execution on the compute system — and `demoted` — jobs that were previously `on-deck` but no longer are.

Once the scheduler has found a new job that has changed to the `RESOURCES` state the job is now considered to be `on-deck` by the RSA Scheduler and a set of RSA nodes is allocated to the job in proportion to the number of compute nodes requested. A fixed ratio of compute nodes to each RSA node is used here and is adjusted to match the system scale. The allocated RSA nodes are removed from the list of free nodes and marked in the scheduler's state files as belonging to that job. If the required number of RSA nodes is not available the job is skipped over in the current scheduling iteration. The expectation is that a later pass of the RSA Scheduler will be able to allocate nodes before the job begins execution. The `rsa-state` transitions for `on-deck` jobs are shown in Figure 2.

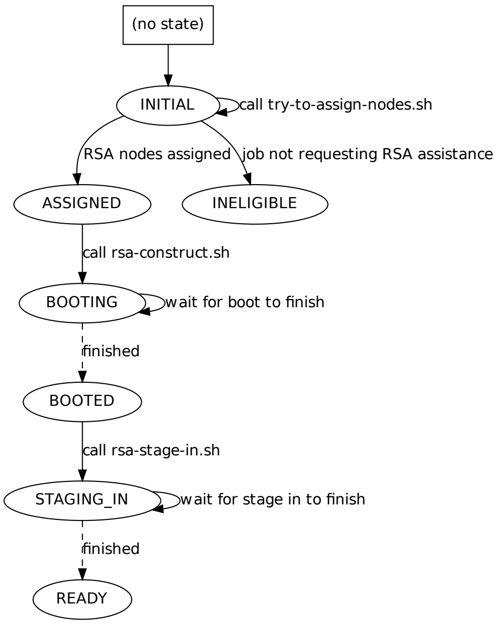

Figure 2: RSA state change diagram for `On-Deck` jobs. For this and the following three figures the boxed nodes denote the ideal starting and ending states and the dashed lines indicate state transitions that are triggered by external call-outs.

A separate process is then started to take the allocated nodes and construct the parallel RAMDISK for use with the job. Once this step completes and the RSA is marked as being ready for use, the next RSA Scheduler iteration will start the data stage-in process.

Once the job data has been successfully staged in, the RSA Scheduler can attempt to *lock-in* the job scheduled and prevent it from being preempted. This is accomplished by setting a quality-of-service flag for the job in SLURM, making it highly unlikely that SLURM would demote this job and force us to tear-down the RSA and reallocate it.

Before the job has started on the compute system it may be rescheduled and the RSA allocation would then need to be revoked. This `demoted` job is defined as any job changing status in SLURM from pending waiting on `RESOURCES` to pending waiting on `PRIORITY`. Once a job has been `demoted` the data staging process is stopped, the associated RAMDISK is destroyed and the allocated RSA nodes are released. State transitions for this state are shown in Figure 3. Note that in all of these state transition diagrams the `rsa-state` may have been set by the scheduler in a different `job-state` and these transitions must be handled properly in the current `rsa-state`.

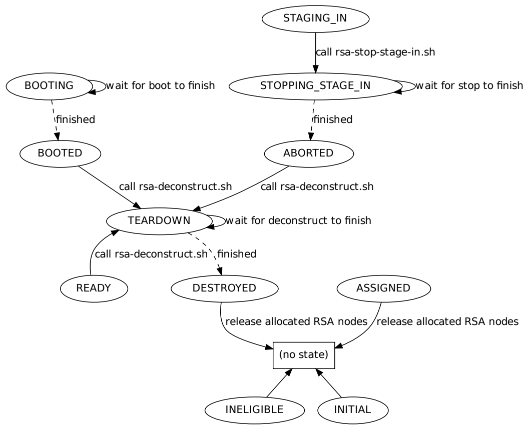

Figure 3: RSA state change diagram for `Demoted` jobs.

There is a further complication — it is possible that a job would jump from pending on `RESOURCES` to `RUNNING` on the compute system before the data stage-in process has completed. In this case, rather than delay the start of job execution until the stage-in completes, the RAMDISK is instead ignored and the job will read data in directly from the disk storage. It is expected in this case that the time taken to complete staging data in to the RAMDISK would be approximately the same as that necessary to read it directly from the compute system itself, and thus, the RSA is ignored for the sake of expediency[2].

3.2 Job Execution

If the RSA has successfully staged the data for the job in to the RSA nodes, the RSA space must be made available to the compute job. Directly prior to execution beginning a script, called from SLURM's *Prolog* script, checks the `rsa-state`, and (if the stage-in has finished) it *bind mounts* the RSA directory over the original part of the filesystem on the I/O nodes assigned to the job.

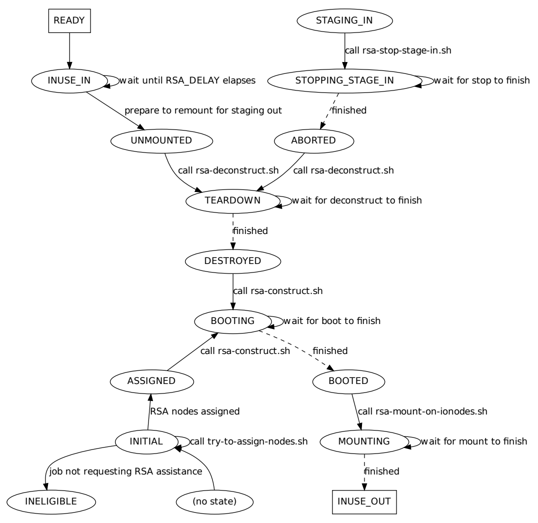

Figure 4: RSA state change diagram for `Running` jobs.

By using a bind mount the RSA directory is made to appear to be at the original filesystem location of the data stage-in directory. The compute job does not need to know the RSA status. If the RSA was unable to stage in all of the data for the job before the job started or was not allocated RSA nodes before launching, the job would instead be reading data in from the disk filesystem directly — albeit at a reduced speed.

Some time in to the job launch, the RAMDISK is reset to empty it and prepare it to receive output data. The specific state transitions are shown in Figure 4. In brief, the following steps are taken:

- The bind mounts on the I/O nodes are released. Note that if the job needs to read additional data in from the `RSA_DATA_IN directory` it would be reading from the disk filesystem directly, as the underlying disk storage is then exposed to the job directly[3].
- The RAMDISK is destroyed then recreated again from scratch, providing a new empty RAMDISK.
- The fresh RAMDISK is bind-mounted over the `RSA_DATA_OUT` directory on the I/O nodes assigned to the job.

Once this completes, the RSA Scheduler will make no further changes for this job until the job moves to the `finished` state.

[2] A further extension to the RSA Scheduler would add an option for the job to control this behavior. It is possible that having the data staging complete would be preferable for other reasons, especially if the stage-in process was responsible for unpacking and pre-processing the data rather than simply proving an accelerated cache of the files stored on disk. Alternatively, the stage-in script could be stopped, and the Linux *union* mount could be used on the I/O nodes to make the current contents of the RAMDISK accessible alongside the full contents of the original directory — this would provide faster access to data that has been staged in, while files that were not staged would remain directly accessible from the disk storage system.

[3] The mounts will not be released until any open files have been closed. If the job does not close out the input files until the end of the job, the RSA will not be used to stage data out as this transition to prepare the RSA to stage-out will not complete until after the compute job finishes.

3.3 Data Staging Out and RSA Tear-Down

Once the job completes, the SLURM *Epilog* routine will unmount the RAMDISK from the I/O nodes and update the RSA Scheduler's job state information. The RSA Scheduler then handles the `finished` state transitions as shown in Figure 5. Briefly, the RSA Scheduler:

- Removes the bind mounts from the I/O nodes.
- Stages data back out to the disk storage systems. If requested, a custom post-processing script can be run here. This is controlled by the `RSA_POSTPROCESS` variable in the SLURM job file. Otherwise, the default stage-out script will copy data from the RAMDISK back to the `RSA_DATA_OUT` directory in the disk storage.
- Destroys the RAMDISK filesystem and releases the RSA nodes back to the RSA Scheduler for allocation to another job.

3.4 Data Staging

The data staging steps, both for moving data in before job execution and for pushing data back to the disk storage system after job completion, are designed to be flexible and allow for customization by the end-user.

The default data flow implemented by the RSA Scheduler is to read data in from the user specified `RSA_DATA_IN` directory to the RAMDISK. At a predetermined time the RAMDISK is reset to receive output data. After job completion data is copied from the RAMDISK back to the user specified `RSA_DATA_OUT` directory on the disk filesystem.

The user can provide `RSA_PREPROCESS` or `RSA_POSTPROCESS` scripts to override this default behavior. These scripts are executed under their user account through use of the `sudo` command to ensure that the underlying filesystem security is maintained. These scripts are able to perform more complex operations than simply copying the data back and forth.

Scientific applications that structure their results as a set of many independent files, usually one file per process thread [8, 9] could potentially see significant performance gains using these custom scripts. Applications have historically used this file-per-process pattern to simplify their I/O behavior as each compute thread independently writes output data taking full advantage of all of the I/O nodes associated with that job. The downside to this approach is that it tends to lead to poor filesystem performance due to bottlenecks writing out file metadata [2, 24]. As an alternative to this file-per-process model, libraries such as Parallel HDF5 [6] and Parallel netCDF [13] aim to improve performance for scientific data sets by coordinating access to a single shared file. The trade-off with these approaches is that they require modification to the application itself.

The RSA system can then improve these applications without modification by using a custom data staging script. The application can write out these many-small-files to the RAMDISK and the stage-out script can aggregate them together into a single file. One simple mechanism to accomplish this is to use the *tar* command to package them into one larger file. Advanced cases could use the RSA nodes to post-process the data and distill it to a form more convenient for the end user. Compression could also be used on the results in preparation for transfer outside of the HPC center.

4. PROOF-OF-CONCEPT SYSTEM

The RAMDISK Storage Architecture was developed to meet RPI's requirements for a next-generation supercomputing facility [3]. The full-scale design is 4-Terabyte RSA cluster (consisting of 32 nodes, each with 128GB of 1333 MHz DDR3 DRAM) connected over Infiniband to a 512-node IBM Blue Gene/Q.

The proof-of-concept implementation is designed to match the operating environment of the full-scale system as closely as possible. An IBM Blue Gene/L acts as a stand-in for the proposed Blue Gene/Q system. Architectural similarities mean that integration work done in this proof-of-concept system should translate to the proposed Blue Gene/Q system and its associated management interfaces with only slight adjustments. The stand-in for the RSA cluster is constructed from similar hardware to the target environment.

The RAMDISK Storage Accelerator in the proof-of-concept system was implemented by borrowing 16 nodes of the Scientific Computation Research Center's *Hydra* cluster. Each node in the Hydra cluster contains a 2.3GHz, 8-core AMD Opteron processor and 32 GB of 1333MHz ECC DDR3 memory and is connected to a Gigabit Ethernet network. The cluster nodes run Debian GNU/Linux 6.0 with a custom 2.6.37 Linux kernel and use PVFS [12] version 2.8.2 to construct the RAMDISKs.

The compute system used for testing is the RPI *SUR Blue Gene/L* , consisting of 1024 compute nodes and 32 I/O nodes. PVFS version 2.8.2 was installed on both the I/O nodes in the system, as well as the frontend node, allowing both to directly access the PVFS-based RAMDISKs exported by the Hydra cluster.

The systems were linked together by running a single Gigabit Ethernet link between the Gigabit Ethernet switch in the Hydra cluster and the functional network's Gigabit Ethernet fabric in the SUR Blue Gene/L. Due to this 1-Gigabit Ethernet bottleneck between the RSA cluster and the compute system, results in the form of I/O performance improvements were not specifically sought for. Due to this bottleneck, the proof-of-concept system cannot demonstrate the order-of-magnitude performance advantages expected from the RSA system in the full-scale environment. Instead, the primary purpose of the proof-of-concept system is to demonstrate that the scheduling mechanisms function properly, that dynamic creation of the necessary RAMDISKs can be managed, that the correct set of I/O nodes are able to access the correct RAMDISKs and that data stage-in and stage-out mechanisms behave as designed.

5. PERFORMANCE RESULTS

A series of test jobs were run to demonstrate performance improvements from using the RSA on the proof-of-concept system. An important caveat here is that any performance results are influenced by the 1-Gigabit network bottleneck between the RSA nodes and the I/O nodes and that this is an expected limitation of the proof-of-concept environment.

5.1 Results for a Single Compute Thread and Single File

A first attempt at comparing performance between the GPFS system and the RSA in the prototype demonstrated only a minor improvement. The results are shown in Table 1. The test case was configured to write out a 2GB

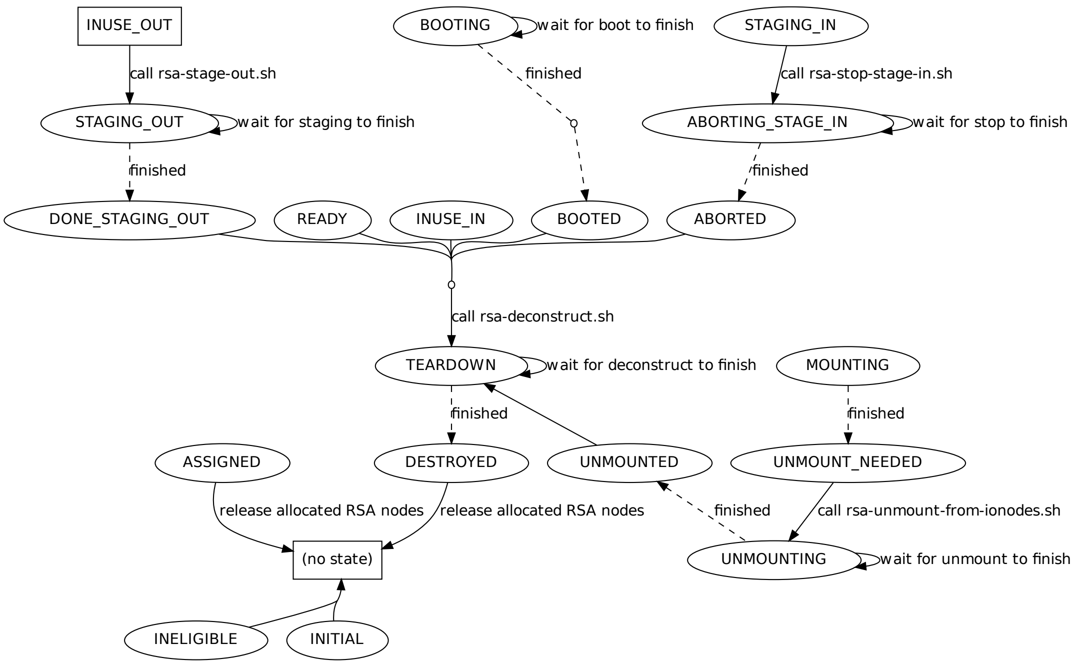

Figure 5: RSA state change diagram for `Finished` jobs.

file through a single compute thread. Only a minor difference between writing results out via the RSA, versus directly to the GPFS system can be seen — a 13% speedup, a far cry from the order-of-magnitude improvement expected from the full-scale system. In both the RSA and GPFS cases the peak speed attained by the single compute thread is close to 50MB/s. Both the RSA and GPFS systems should be capable of better performance than this. On further investigation prior results were found demonstrating that a single I/O node's network performance (and thus its networked filesystem performance) on a Blue Gene/L system is around 50MB/s [26]. Thus, this result primarily demonstrates this same bottleneck.

Table 1: Comparison of output performance of GPFS disk storage vs. RSA, single thread writing. Times are in seconds.

	No RSA	With RSA	
	Output Time	Output Time	Stage-Out Time
Run 1	46.60	41.19	54
Run 2	47.69	41.44	48
Run 3	46.35	41.27	54
Mean	46.88	41.30	52

5.2 Results for a Single-File-Per-Process over 1024 Nodes

A second approach to demonstrating I/O differences between the systems is to run the test case under One-File-Per-Process mode. Table 2 shows the consolidated results for three runs in three different modes of operation. The first case is for writing results directly to GPFS and the second and third cases are for writing to the RAMDISK instead. The second and third cases differ only in the method used to stage data back to disk.

Writing out the 2048 files from the 1024-node job to the GPFS filesystem directly takes an average of 1109 seconds, or over 18 minutes, while the same data written to the RSA instead takes only 37 seconds — a 2800% speedup. This slow performance from GPFS is due to lock contention on the output directory. This is a known limitation of GPFS [7].

The data written to the RAMDISK needs to be staged-out to the GPFS system. Results for two methods for staging data back to the GPFS system are shown. The first method uses the default stage-out script, which uses the `rsync` command, to transfer the data back to GPFS. This stage-out step directly shows the performance gains achievable by avoiding contention on GPFS as the resulting files after the stage-out has finished are the same as they would be if written directly to GPFS by the compute job. This gain is due to the stage-out being handled by a single process writing data sequentially, as opposed to 2048 compute threads simultaneously competing for access, which avoids metadata lock contention in GPFS.

The second stage-out method inserts a custom `RSA_STAGE_OUT` script. This script is configured to use the `tar` command to combine the separate files into a single output file (with no compression). This clearly shows a performance gain versus the default `rsync` method — storing the results in one large file is definitely advantageous. Due to the asynchronous nature of the RSA the time taken to stage back to disk is relatively unimportant as long as the RSA nodes are free before the next job would need to use them to stage data in.

Table 2: Comparison of output performance of GPFS disk storage vs. RSA, 1024 nodes (2048 processes) in file-per-process, with normal and custom post-processing scripts. Times are in seconds.

	No RSA	RSA With Default Stage-Out		RSA with Custom Stage-Out	
	Output	Output	Stage-Out	Output	Stage-Out
Run 1	1158.81	36.11	222	35.76	179
Run 2	1193.92	36.25	227	36.25	178
Run 3	976.84	35.26	224	35.97	181
Mean	1109.86	35.87	224	36.43	179

While these results clearly show a deficiency in the GPFS implementation, it must be stressed that the One-File-Per-Process mode is a common one among HPC applications [8, 9, 11, 19], and this is clearly an instance where the RSA would be particularly beneficial.

5.3 IOR Benchmark Results

The IOR synthetic filesystem benchmark [21] was run against the PVFS-based RSA RAMDISK and the GPFS filesystem to compare relative performance. IOR was configured to run four write-out and read-in passes against both, with a 1MB file per process (2048 files). Results as an average of the four runs on each system are shown in Table 3. Notably, the IOR benchmark times the initial delay in opening each file. This value directly demonstrates the overhead in GPFS for file creation.

Table 3: Consolidated Results from the IOR benchmark on both RAMDISK and GPFS filesystems, over 1024 nodes.

	RSA	GPFS
Write Time (seconds)	204.16	573.28
File Open Delay (seconds)	0.43	83.27
Read Time (seconds)	187.48	289.12
Write, MBytes/sec	100.31	36.18
Read, MBytes/sec	109.24	70.89

The RSA again demonstrates its value with a 180% performance improvement when writing results out versus the GPFS filesystem. The read performance for the two systems is much closer here and demonstrates only a 54% improvement for the RSA. Especially telling here is the delay in opening files for writing out — GPFS needs an additional 80 seconds compared to the RSA. This delay is not reflected in the write throughput performance number given before. If it were factored in the RSA would show a 220% performance gain instead.

Also specifically of interest is the 100MBytes/sec value that the RSA achieves on both read and write performance — this corresponds to the maximum performance possible given the Gigabit Ethernet bottleneck between the systems. To demonstrate this the *iperf* [22] benchmark was run between the frontend node on the SUR Blue Gene/L and a Hydra node to quantify the maximum performance possible between these systems. A peak bandwidth between them of 943Mbit/sec was observed, or 117.9MBytes/sec. The iperf result correlates to the maximum speed possible over a Gigabit Ethernet link [5] and the read performance result achieved by the RSA is then within 8% of this maximum value. The RSA results are certainly affected by this network bottleneck.

6. CONCLUSION

The *RAMDISK Storage Architecture* presents a novel method of handling the growing divide between I/O throughput and compute system power on large-scale HPC systems. Dedicated I/O resources in the form of parallel RAMDISKs — virtual storage space backed by DRAM on individual *RSA nodes* and aggregated together using a parallel filesystem — are assigned on-demand to jobs on the compute system. Asynchronously staging data in and out of these RAMDISKs provides a mechanism to support higher throughput on the compute system, as jobs no longer sit idle on the compute system waiting for data to be loaded-in or written-out to a comparatively slow persistent disk storage systems. Instead each job makes use of the higher-performance RAMDISK assigned to it to read initial datasets in and write results out.

The *RSA Scheduler* implements the required asynchronous data staging and RSA system management mechanisms by providing an additional scheduling layer built around the SLURM job scheduler. The RSA Scheduler is implemented as an asynchronous state-transition machine such that the core scheduling duties are handled in a minimal time and external operations such as RAMDISK construction and deconstruction, data staging, and node management do not impact the core scheduling mechanism.

A proof-of-concept system has been demonstrated with the prototype RSA Scheduler in operation and demonstrates the viability of the asynchronous staging model. Performance results, including a 2800% improvement in one specific instance, are shown for running with and without use of the RSA on a proof-of-concept system.

7. REFERENCES

[1] Abbasi, H., Wolf, M., Eisenhauer, G., Klasky, S., Schwan, K., and Zheng, F. DataStager: scalable data staging services for petascale applications. In *Proceedings of the 18th ACM International Symposium on High Performance Distributed Computing* (Garching, Germany, June 2009), ACM, pp. 39–48.

[2] Biardzki, C., and Ludwig, T. Analyzing Metadata Performance in Distributed File Systems. In *Parallel Computing Technologies (10th PaCT'09)*, V. Malyshkin, Ed., vol. 5698 of *Lecture Notes in Computer Science (LNCS)*. Springer-Verlag (New York), Novosibirsk, Russia, Aug.-Sept. 2009, pp. 8–18.

[3] Carothers, C., Shephard, M., Myers, J., Zhang, L., and Fox, P. MRI: Acquisition of a Balanced Environment for Simulation. Jan. 2011. URL: `http://nsf.gov/awardsearch/showAward.do?AwardNumber=1126125`.

[4] Cope, J., Liu, N., Lang, S., Carns, P., Carothers, C., and Ross, R. CODES: Enabling Co-design of Multilayer Exascale Storage Architectures. In *Proceedings of the Workshop on Emerging Supercomputing Technologies 2011* (2011), ACM.

[5] Dykstra, P. Protocol Overhead. URL: `http://sd.wareonearth.com/~phil/net/overhead/`.

[6] Folk, M., Cheng, A., and Yates, K. HDF5: A file format and i/o library for high performance computing applications. In *Supercomputing'99 Conference Proceedings* (Portland, OR, Nov. 1999), ACM/IEEE.

[7] Frings, W., and Hennecke, M. A system level view of Petascale I/O on IBM Blue Gene/P. *Computer Science - Research and Development 26* (2011), 275–283.

[8] Fu, J., Liu, N., Sahni, O., Jansen, K. E., Shephard, M. S., and Carothers, C. D. Scalable parallel I/O alternatives for massively parallel partitioned solver systems. In *IPDPS Workshops* (2010), IEEE, pp. 1–8.

[9] Fu, J., Min, M., Latham, R., and Carothers, C. Parallel I/O Performance for Application-Level Checkpointing on the Blue Gene/P System. In *Workshop on Interfaces and Abstractions for Scientific Data Storage* (Austin, TX, Sept. 2011), IEEE.

[10] Iskra, K., Romein, J. W., Yoshii, K., and Beckman, P. ZOID: I/O-forwarding infrastructure for petascale architectures. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming* (New York, NY, USA, 2008), PPoPP '08, ACM, pp. 153–162.

[11] Lang, S., Carns, P. H., Latham, R., Ross, R. B., Harms, K., and Allcock, W. E. I/O performance challenges at leadership scale. In *SC'09 Conference Proceedings* (Portland, OR, USA, Nov. 2009), ACM/IEEE.

[12] Latham, R., Miller, N., Ross, R., and Carns, P. A Next-Generation Parallel File System for Linux Clusters. *LinuxWorld 2*, 1 (Jan. 2004).

[13] Li, J., keng Liao, W., Choudhary, A., Ross, R., Thakur, R., Gropp, W., Latham, R., Siegel, A., Gallagher, B., and Zingale, M. Parallel netCDF: A High-Performance Scientific I/O Interface. In *SC2003 Conference Proceedings* (Phoenix, AZ, Nov. 2003), ACM/IEEE.

[14] Lofstead, J. F., Zheng, F., Liu, Q., Klasky, S., Oldfield, R., Kordenbrock, T., Schwan, K., and Wolf, M. Managing Variability in the IO Performance of Petascale Storage Systems. In *SC'10 Conference Proceedings* (New Orleans, LA, USA, 2010), ACM/IEEE.

[15] Moody, A., Bronevetsky, G., Mohror, K., and de Supinski, B. R. Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System. In *SC'10 Conference Proceedings* (New Orleans, LA, USA, Nov. 2010), ACM/IEEE.

[16] Nowoczynski, P., Stone, N., Yanovich, J., and Sommerfield, J. Zest: Checkpoint Storage System for Large Supercomputers. In *SC'08, 3rd Petascale Data Storage Workshop* (Austin, TX, Nov. 2008), ACM/IEEE.

[17] Ohta, K., Kimpe, D., Cope, J., Iskra, K., Ross, R., and Ishikawa, Y. Optimization Techniques at the I/O Forwarding Layer. In *IEEE International Conference on Cluster Computing 2010* (Los Alamitos, CA, USA, 2010), IEEE Computer Society, pp. 312–321.

[18] Ousterhout, J. K., Agrawal, P., Erickson, D., Kozyrakis, C., Leverich, J., Mazières, D., Mitra, S., Narayanan, A., Parulkar, G., Rosenblum, M., Rumble, S. M., Stratmann, E., and Stutsman, R. The Case for RAMClouds: Scalable High-Performance Storage Entirely in DRAM. *SIGOPS Operating Systems Review 43*, 4 (Dec. 2009), 92–105.

[19] Polte, M., Simsa, J., Tantisiriroj, W., and Gibson, G. Fast Log-based Concurrent Writing of Checkpoints. In *SC'08, 3rd Petascale Data Storage Workshop* (Austin, TX, Nov. 2008), ACM/IEEE.

[20] Raicu, I., Foster, I. T., and Beckman, P. Making a case for distributed file systems at Exascale. In *Proceedings of the Third International Workshop on Large-Scale System and Application Performance* (New York, NY, USA, 2011), LSAP '11, ACM, pp. 11–18.

[21] Scalable I/O Project, LLNL. IOR Benchmark. URL: `http://sourceforge.net/projects/ior-sio/`.

[22] Tirumala, A. End-to-End Bandwidth Measurement Using Iperf. In *SC'2001 Conference Proceedings* (Denver, CO, USA, Nov. 2001), ACM/IEEE.

[23] Vishwanath, V., Hereld, M., Iskra, K., Kimpe, D., Morozov, V., Papka, M. E., Ross, R., and Yoshii, K. Accelerating I/O Forwarding in IBM Blue Gene/P Systems. In *SC'10 Conference Proceedings* (New Orleans, LA, USA, Nov. 2010), ACM/IEEE.

[24] Weil, S. A., Pollack, K. T., Brandt, S. A., and Miller, E. L. Dynamic Metadata Management for Petabyte-Scale File Systems. In *SC'2004 Conference Proceedings* (Pittsburgh, PA, USA, Nov. 2004), ACM/IEEE.

[25] Wickberg, T. The RAMDISK Storage Accelerator - A Method Of Accelerating I/O Performance On HPC Systems Using RAMDISKs. Master's thesis, Rensselaer Polytechnic Institute, Troy, NY, USA, 2011.

[26] Yu, H., Sahoo, R. K., Howson, C., Almasi, G., Castaños, J. G., Gupta, M., Moreira, J. E., Parker, J. J., Engelsiepen, T., Ross, R. B., Thakur, R., Latham, R., and Gropp, W. D. High performance file I/O for the Blue Gene/L supercomputer. In *High Performance Computer Architecture* (2006), IEEE Computer Society, pp. 187–196.