THE RAMDISK STORAGE ACCELERATOR — A METHOD OF ACCELERATING I/O PERFORMANCE ON HPC SYSTEMS USING RAMDISKS

By

Timothy B. Wickberg

A Thesis Submitted to the Graduate

Faculty of Rensselaer Polytechnic Institute

in Partial Fulfillment of the

Requirements for the Degree of

MASTER OF SCIENCE

Major Subject: COMPUTER SCIENCE

Approved:

_____

Christopher D. Carothers, Thesis Adviser

Rensselaer Polytechnic Institute
Troy, New York

November 2011
(For Graduation December 2011)

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGMENTS

Many thanks to Professor Chris Carothers, Professor Mark Shephard, and Professor Wolf von Maltzahn for their guidance and thoughful advice over the years.

Thanks to Adam Todorski for the discussions and initial testing a few years ago that led to this work, and for proof-reading this thesis. Footnote 13 is entirely his idea.

Thanks to Keegan Trott for proof-reading this; that semicolon is for him.

And of course, thanks to my family and friends for their love and support.

ABSTRACT

I/O performance in large-scale HPC systems has failed to keep 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; one that leverages the high-throughput and decreasing cost of DRAM, while providing an application-transparent method for pre-staging input data and committing 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 then 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 thus provides for very-high-speed, low-latency access that is dedicated to a specific job; the 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.

To support this asynchronous data-staging model requires an method of scheduling this new capability alongside that of the traditional task of scheduling compute resource access for each job. A proof-of-concept implementation based on the SLURM job scheduler is presented, and demonstrates an operational 16-node RSA system connected to a 1024-node IBM Blue Gene/L.

This thesis presents a case in favor of this RAMDISK Storage Architecture along with specific work done to implement a proof-of-concept system demonstrating the feasibility of this mode of operation.

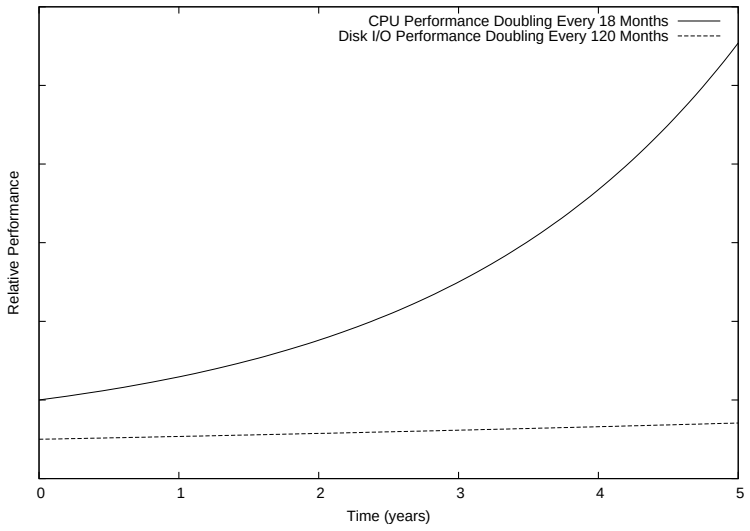
CHAPTER 1
Introduction and Historical Review

1.1 I/O Performance in HPC Systems

Filesystem I/O performance in large-scale HPC systems has failed to keep pace with improvements in computational performance [10, 27, 48, 51]. As HPC applications continue to evolve their 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 is spent waiting for I/O to finish [30]. 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, 2, 6, 12, 41, 49].

Figure 1.1 illustrates the growing divide. Moore's Law predicts that CPU performance will double every 18 months [44]. However I/O throughput for a single

Figure 1.1: The I/O Gap — CPU Performance Versus Disk Throughput Over Time



disk has only doubled once every 10 years or so [18]. As HPC systems continue to evolve disk throughput will continue to lag behind unless an exponentially-growing number of disk drives are added to the storage systems, or a radically different approach to disk storage systems is taken.

We believe that an opportunity exists to introduce a new system layer — one that is able to decouple the disk storage system from the compute system, while behaving in a method that is application-independent and transparent. The proposed solution is the introduction of a new component to the HPC systems architecture — the *RAMDISK Storage Accelerator* (*RSA*).

By aggregating the DRAM available in a number of commodity servers, each referred to as RSA nodes, a RAMDISK can be constructed using a parallel filesystem such as Ceph [52], PVFS [28], GPFS [45] or Lustre [8]. This parallel RAMDISK can be presented to the compute system as if it were a traditional disk-backed filesystem. The RAMDISK thus 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* is introduced to implement this asynchronous staging and manage 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. These mechanisms are transparent to the application itself — no modifications need to be made to the application to take advantage of the RSA.

A proof-of-concept implementation of this RSA Scheduler is discussed and demonstrated in operation as a proof-of-concept system. It uses currently available systems on the RPI campus — a 1024-node IBM Blue Gene/L and an HPC cluster

acting as a 16-node RAMDISK Storage Accelerator.

1.2 Historical Review

The use of DRAM as a caching layer for filesystem access is well established. There has been a growing interest in methods of accelerating application storage, especially databases and key-value stores, through use of applications designed to operate exclusively in-memory. In particular the introduction of *memcached* [16] to the web application community has been met with favorable results. Memcached is a key-value store implemented exclusively in memory, designed for high-speed data caching. Applications are then expected to implement their own persistent storage model on top of this, generally committing data back to disk storage systems on demand. This works especially well in read-intensive environments.

The introduction of Solid-State Disks (SSDs) to the HPC environment has also provided an alternative method of accelerating storage [3, 10, 23]. SSDs provide considerably higher throughput and much better random I/O performance than traditional hard disks. In particular, they have been successfully applied towards small-file storage systems. Filesystem metadata has received specific attention as it has been a major bottleneck in large-scale systems [3, 20, 27, 54], and SSDs provide a direct cost-effective improvement there. The trade-off is in their substantially higher cost — the Petabyte storage systems that accompany current leadership-class HPC systems would be prohibitively expensive to implement on current SSD technology [23]. Even so, the *Gordon / Dash* system explores the use of SSDs to accelerate specific data-intensive applications [23, 34] for specific workloads. One result of shows a comparison between in-memory performance (similar to our RAMDISKs), SSDs and disks, but does not extend the result to the potential for a dedicated system such as in the RSA.

A recent development using DRAM to provide a parallel filesystem entitled *RAMCloud* [37] first appeared in 2009. The RAMCloud implementation is designed to hold all filesystem data in memory, and commits a copy of the data to storage on-demand. A key divergence between RAMCloud and the proposed RSA is that RAMCloud, by design, provides a dedicated persistent filesystem. It accomplishes

this by committing data written back to underlying disk storage on each node and rebuilding each node by reading data back into DRAM if the node has been restarted. The RAMDISK approach used by the RSA intentionally discards data on restart, and requires an explicit commit step to move data back to a persistent storage layer. The RSA approach is meant to supplement persistent storage; the RAMCloud approach intends to replace it wholesale and relies on relatively small amounts of disk storage for persistence. Additionally, the RAMCloud approach is implemented as a key-value store similar to memcached, while the RAMDISK in the RSA is implemented using a conventional parallel filesystem, PVFS [28], and can be used by current HPC applications without modification.

Asynchronous data staging, in a similar manner to what is implemented for the RSA, is also discussed in relation to different classes of disk storage as the *Zest* system [35]. 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 [51]. The *Maxperf* system would provide a mechanism to allocate cache space on the filesystem storage servers themselves to specific jobs on demand [39]. 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, that system relies on application modification to offload the I/O duties to the DataStager systems, whereas the RSA presents a standard POSIX file interface. *IOFSL* [36] and *ZOID* [24] 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 [32]. 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.

CHAPTER 2
RAMDISK Storage Accelerator Architecture

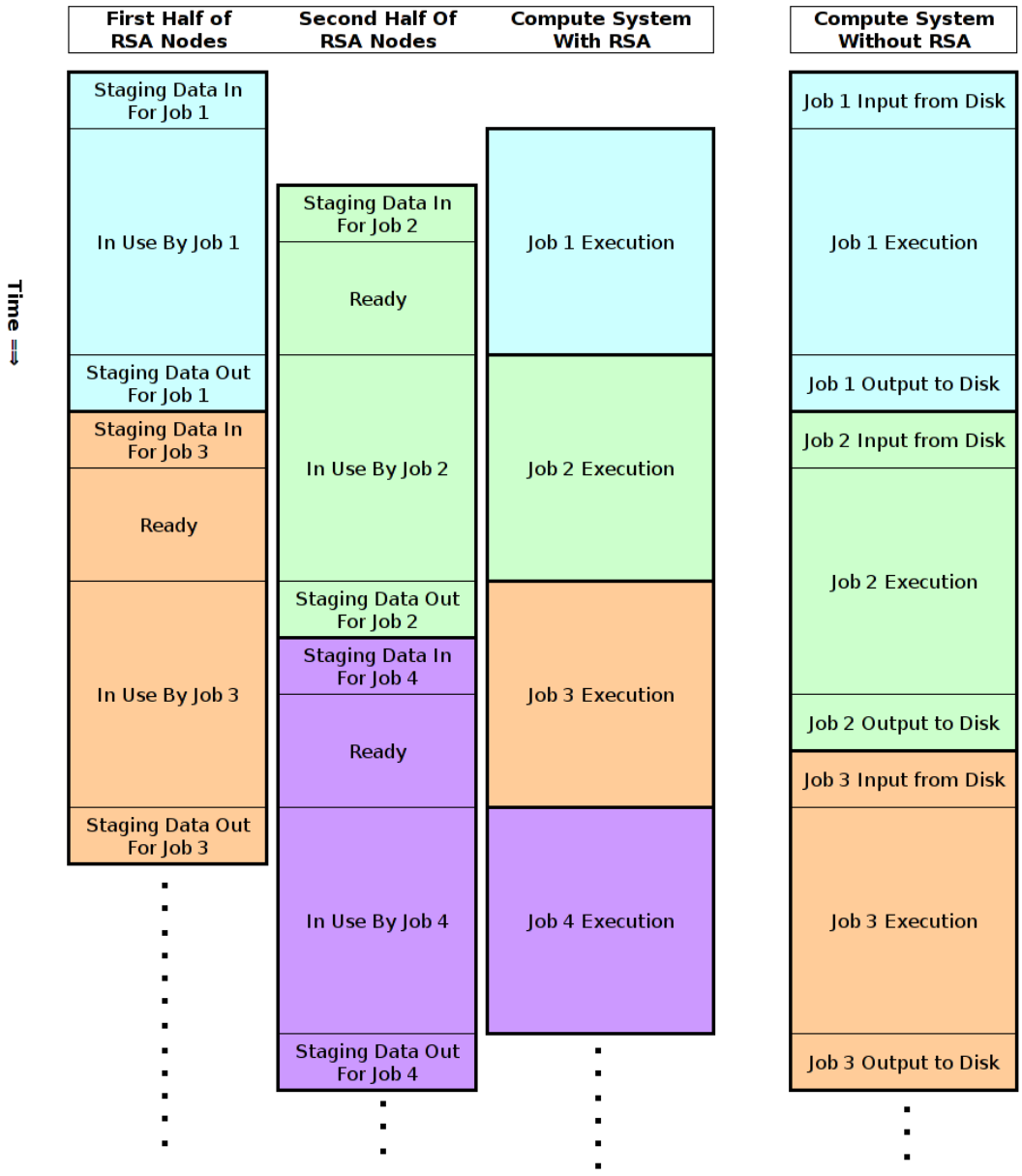
2.1 Design of the RAMDISK Storage Accelerator

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 will begin execution on the compute system. Once execution begins the job is able to access data from the RAMDISK at a much higher speed than it would out of the persistent 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 is able to release its allocation on the compute system, and the RSA will stage data out of the RAMDISK back to persistent storage asynchronously.

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 I/O 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. Figure 2.1 shows an idealized scenario demonstrating this capability. In this figure the compute system is continuously occupied by running jobs — jobs that are able to take advantage of the high-speed I/O provided by the RAMDISK — while the RSA nodes handle staging data in and out of the system asynchronously. The fourth column shows how the same jobs would behave without the advantages of the RSA, while the underlying computation steps take the same amount of time, the increased time spent reading data in and writing results out is shown and reflects a decreased throughput for 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

Figure 2.1: Simplified and idealized RSA Scheduler logic, showing an ideal schedule for a series of full-compute-system jobs running alongside the RSA, as well as the same system running without the RSA. The time dimension is shown heavily compressed and unscaled to demonstrate the behavior, no specific performance improvement is implied here.



parallel filesystem allows us to aggregate 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 asynchronously from job execution, this can occur at a much lower rate than would be traditionally required on the compute system itself.

Once the job starts, execution on the compute system can read its data in from the RSA at a much higher speed than it would from the disk storage. Once this has completed, the RSA is able to discard the data (as it remains in the persistent disk storage system) and reset itself to prepare to receive data 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, it is then free to start the next job — it does not need to sit idle, waiting for data to be pushed out to disk storage.

The RSA is then used to stage data back to the disk storage system. As with the initial data staged in at the job start, the performance of the disk storage system no longer directly impacts the throughput of the compute system itself. Additionally, 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[1]. 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 [3, 7, 54].

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[2]. Traditionally all compute jobs, outside file

[1] A common occurrence with application codes that write one results file per compute thread [35, 38]. As one example, the PHASTA [42] computational fluid-dynamics code is usually run this way [21].

[2] A proposed system — *Maxperf* [39] — would provide a mechanism to allocate storage server cache to allow some control of this.

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 [30, 54]. Additionally, no major parallel filesystems currently provide quality-of-service methods that would allow administrators to control these interactions between systems contending for access[3]. 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[4]. Additionally, as RSA nodes are allocated in direct proportion to job size[5] 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.

2.2 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 [52], Lustre [8], PVFS [28], or GPFS [45].

For this proof-of-concept implementation PVFS is used. While PVFS, Lustre, and GPFS are all currently usable on Blue Gene/L systems, PVFS has one advantage that led to its selection. Specifically, the storage server processes run entirely in user space on each RSA node and does not require custom Linux kernel modules to operate. This makes dynamically creating and destroying the RAMDISKs simpler and more reliable as the storage processes can be easily stopped. Since GPFS and Lustre both run through custom kernel modules, RAMDISKs implemented in either would be difficult to quickly destroy and rebuild[6].

[3] Neither Lustre, GPFS, PVFS, or Ceph provide such a mechanism at present.

[4] This assumes no underlying bottleneck in the HPC center's interconnect.

[5] 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.

[6] Notably, rapid creation and destruction of the filesystem is not usually a design consideration for most filesystems.

On each RSA node a single RAMDISK is created through use of the Linux `rd` pseudo-block device. At boot time the kernel module is loaded by adding a line to the `/etc/modules` file of:

```
rd rd_size=31457280 rd_nr=1
```

This creates a single 30GB RAMDISK on the node that is available as the `/dev/ram0` block device[7]. This block device can be used with any filesystem. The RSA dynamically formats it with the `ext2` filesystem, and then PVFS will run on top of that. PVFS is an object-storage based filesystem and does not directly handle storing data on an underlying block device. Instead, it relies on a separate local filesystem for this — in this case `ext2`. The `ext2` filesystem was chosen for its high performance and simplicity; newer filesystems such as `btrfs` and `ext4` would also work, but present no advantages to the RAMDISK implementation[8].

Some care is taken to dynamically aggregate these RSA nodes together and create the RAMDISK for each job. A program provided with PVFS is used to dynamically create the PVFS configuration file used. Appendix Q is the script used to dynamically construct the PVFS configuration file, the script in Appendix D is used to start PVFS on the RSA nodes, and finally the script in Appendix E destroys the PVFS process on each of the RSA nodes.

[7]The nodes used in the proof-of-concept system each have 32GB of DRAM, the module leaves a 2GB space for the operating system. RAMDISK space is taken directly from the Linux memory manager, and cannot be paged out to swap space. If sufficient space was not left for the operating system it would result in increased use of swap space (slowing down the system waiting on hard disk I/O), and eventually lead to an out-of-memory condition that would crash the RSA node.

[8]Most improvement with these filesystems relates to filesystem integrity through the use of journaling and copy-on-write semantics. The RAMDISK is volatile and would not survive a system restart, so these features would only result in additional overhead for the RAMDISK.

CHAPTER 3
SLURM and the RSA Scheduler

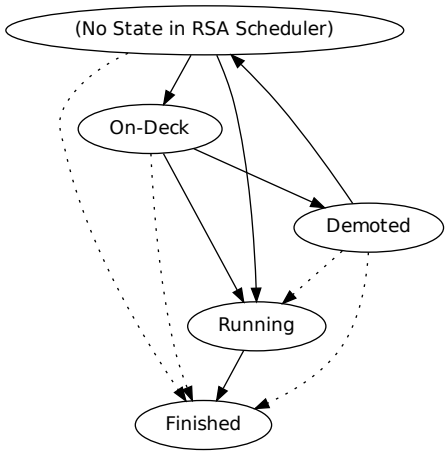
This chapter addresses specific challenges in adapting a common HPC scheduler — SLURM — to managing this proposed RSA system.

For our RAMDISK Storage Accelerator to work efficiently certain stages in a job's lifecycle — including job submission, job execution on the compute system, and cleanup — must be identified by the RSA management system. There are four distinct stages a job can move through that must be handled: `on-deck`, `demoted`, `running` and `finished`. Transitions between these `job-state`s are shown in Figure 3.1. The *RSA Scheduler* is constructed to coordinate these stages to ensure a seamless job execution environment. 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.

For `on-deck` jobs — jobs that are anticipated as being the next to start execution on the compute system — the scheduler must:

- Identify the next pending job on the system that will require RSA resources.

Figure 3.1: Job State changes, showing all possible `job-state` transitions. Dotted lines are unlikely state transitions.



- Allocate an appropriate portion of the RSA cluster to that job, and dynamically construct a parallel RAMDISK on the RSA nodes.
- Stage data in to the new RAMDISK.

Additionally, a state of `demoted` exists for jobs that were in the `on-deck` state, but have been preempted by a higher-priority job in SLURM. For these `demoted` jobs the scheduler must:

- Stop the data staging process, if it is still executing.
- Tear down the RAMDISK, if one was constructed.
- Release the RSA resources the job held, so that they can be used by other jobs in the `on-deck` state.

For jobs that are `running` on the compute system, the RAMDISK must be mounted on the I/O nodes before the computation starts. Then, at some point during execution[9] the scheduler will switch the RAMDISK from providing space for input files to providing space for intermediate checkpoints and results. It accomplishes this by:

- Unmounting the RAMDISK from the I/O nodes, and destroying the current RAMDISK[10].
- Constructing a new RAMDISK on the same RSA nodes.
- Mounting the new RAMDISK on the I/O nodes.

After the job has `finished` on the compute system, the RSA Scheduler needs to:

- Stage data back to the disk storage systems. Or, optionally run a user-provided post-processing script.

[10]This step will not complete if there are still files open by the compute system on the RAMDISK. In this case the RSA will stay in this state until job completion, and will not be used to state data-out. Input files must be closed or located outside of the data stage-in directory for this transition to happen.

[10]This occurs after `RSA_DELAY` seconds of job execution, and may be specified in the job script file, or given a default value by the RSA Scheduler.

- Destroy the RAMDISK used by the process and release the RSA nodes back to the scheduler.

3.1 The SLURM Scheduler

The *Simple Linux Utility for Resource Management* (*SLURM*) scheduler is the HPC scheduler used for our initial proof-of-concept system.

SLURM [55] was originally developed at Lawrence Livermore National Labs and is specifically designed to address issues with job scheduling for large-scale HPC systems. In particular, it provides specific support for the IBM Blue Gene systems and is able to provide dynamic system management on that platform. As our intended target for the full-scale RSA system is the IBM Blue Gene/Q, and SLURM is already providing preliminary support for these yet-unreleased systems [25], it is the best choice for the proof-of-concept system.

SLURM provides for two main modes of job scheduling. The first mode is a traditional *batch scheduler* with weighted job priorities. Jobs are assigned priority values based on certain variables including job size, requested run time, user priority, time the job has spent waiting in the queue, and administratively defined job quality-of-service settings. The job with the highest priority is then run as soon as sufficient resources are available.

The second mode of operation is *backfill* scheduling. In this mode, jobs with the highest priority are tentatively scheduled to run at the earliest opportunity, same as in the batch scheduling mode. If there is a period of time that a number of compute nodes would be idle, while a large job is waiting for sufficient resources to be released, smaller jobs are backfilled in to run immediately as long as the predicted start time for the highest priority job is not delayed excessively. As the backfill scheduler is designed to keep the system running at a higher average usage, and the underlying goal with the RSA is to maximize the overall compute-system usage, it was selected for the proof-of-concept system.

3.2 Job Flow and the RSA Scheduler

This section describes the specific implementation work done for the proof-of-concept system and each step taken to allocate resources to each job.

3.2.1 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 SLURM job queues, and job priorities will be constantly reassessed based on the scheduler's internal state[11], 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 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

[11] SLURM has a set of powerful capabilities to dynamically manage job priorities [26]. These capabilities can cause job priorities to change at any time, and affect which job is chosen to begin execution next.

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 work preparing the RAMDISK begins. 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 3.2.

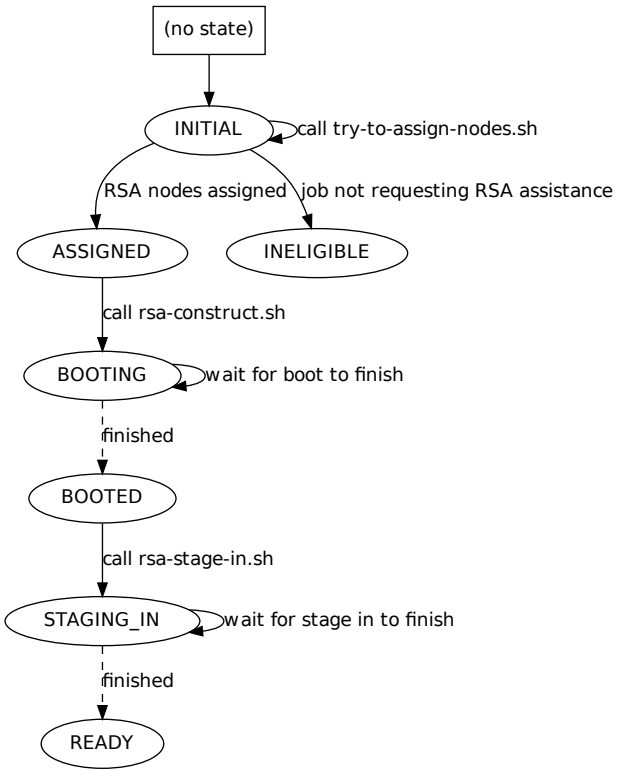
A separate process is then started[12] to take the allocated nodes and construct the parallel RAMDISK for use with the job. Once this step completes and the RSA is marked as being ready for use, the next RSA Scheduler iteration will start the data stage-in process.

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 3.3. Note that in all of these state transition diagrams the `rsa-state` may have been set by the scheduler

[12] Appendix D — **rsa-construct.sh**

Figure 3.2: RSA state change diagram for `On-Deck` jobs. For this figure, and the next three, the boxed nodes denote the ideal starting and ending states, and the dashed lines indicate state transitions that are triggered by external call-outs.

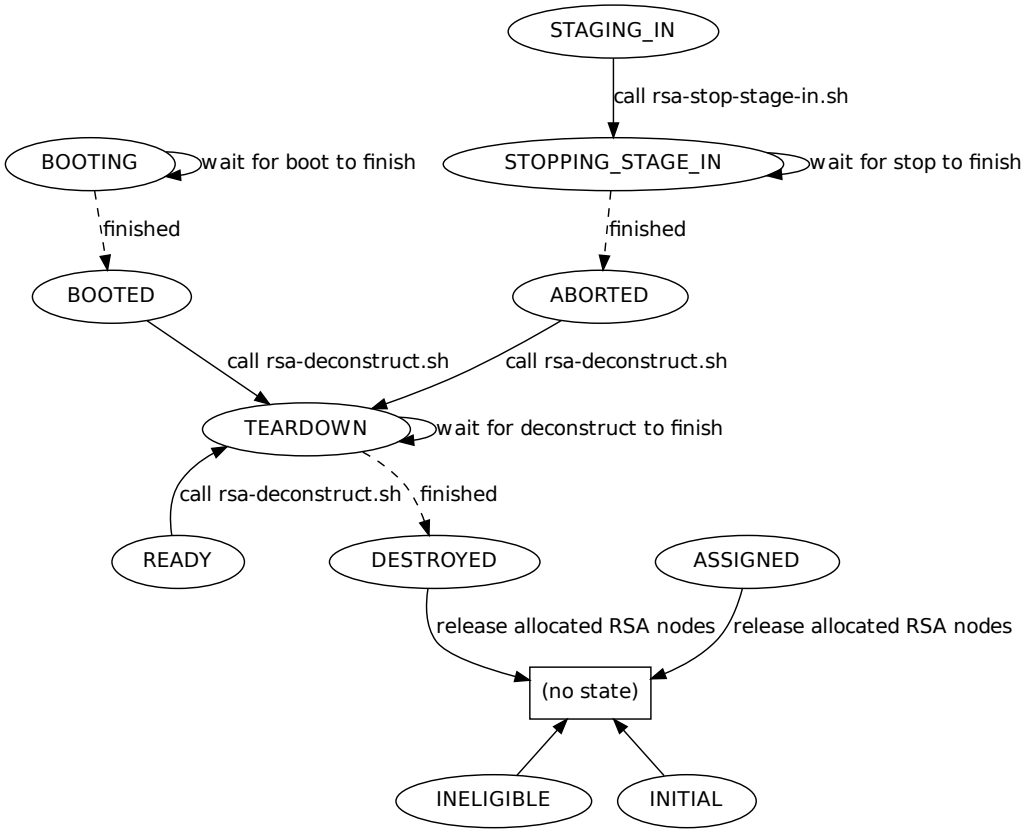


in a different `job-state`, and these transitions must be handled properly in the current `rsa-state`.

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 filesystem. 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[13].

[13]A further extension to the RSA Scheduler would add an option for the job to control this

Figure 3.3: RSA state change diagram for `Demoted` jobs.



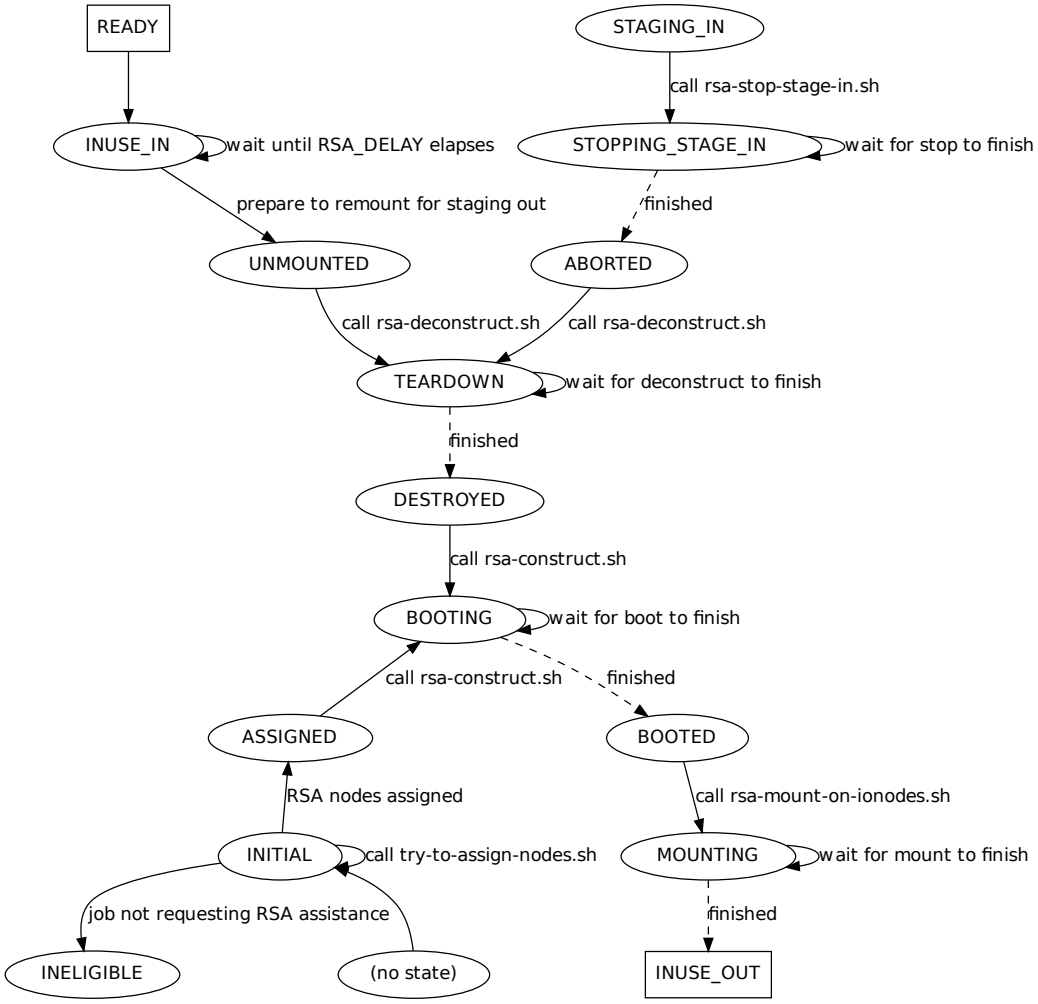
3.2.2 Job Execution

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. A script[14] called through SLURM's *Prolog* script checks the `rsa-state`, and, if the staging has completed, it *bind mounts* the RSA directory over the original part of the filesystem on the I/O nodes assigned to the job.

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.

[14] Appendix K — **job-start.sh**

Figure 3.4: RSA state change diagram for `Running` jobs.



By using a Linux bind mount the RSA directory is made to appear to be at the original 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[15], the RAMDISK is reset to clear it out and

[15]Set to a half-hour by default in the proof-of-concept implementation. This is adjustable by

prepare it to receive output data. The specific state transitions are shown in Figure 3.4. 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[16].
- 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.

3.2.3 Data Staging Out and RSA Tear-Down

Once the job completes, the SLURM *Epilog* routine will, through one of the RSA Scheduler's scripts[17], 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 Figure 3.5. Briefly, the RSA Scheduler:

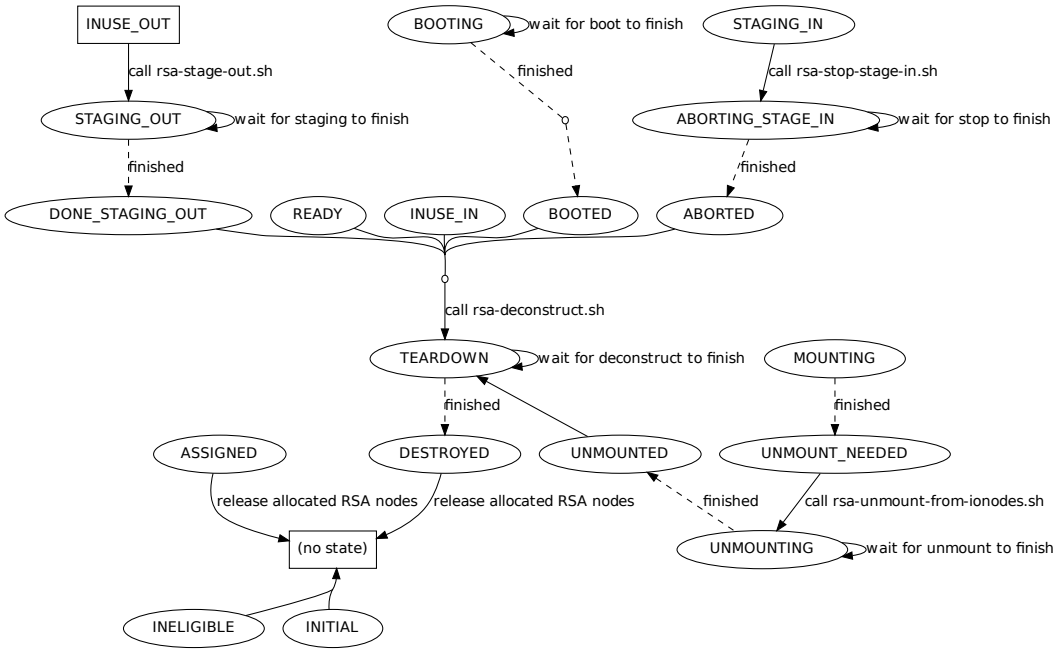
- Removes the bind mounts from the I/O nodes. Note that if the bind mounts were not in place, the RSA Scheduler will skip the data stage-out step and release the RSA nodes immediately.
- Stages data back out to the disk storage systems. If requested, a custom post-processing script can be run here. This 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.

each the job through the `RSA_DELAY` variable.

[16]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.

[17]Appendix N — **job-finish.sh**

Figure 3.5: RSA state change diagram for `Finished` jobs.



- Destroys the RAMDISK filesystem and releases the RSA nodes back to the RSA Scheduler for allocation to another job.

3.2.4 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 have data read in from the `RSA_DATA_IN` directory specified by the job to the RAMDISK. This data is then removed at a predetermined time, and the RAMDISK is reset to receive output data. After job completion data is staged out by directly copying it from the RAMDISK back to the `RSA_DATA_OUT` directory specified by the job on the disk filesystem.

Alternatively, if the user provides either a `RSA_PREPROCESS` or `RSA_POSTPROCESS` script in their job script those are used instead. These scripts are executed under

their user account through use of the `sudo` command to ensure that the underlying filesystem security is maintained. In addition, these scripts are able to perform more complex operations than simply copying the data back and forth.

As an example, there are quite a few scientific applications that structure their results as a set of many independent files, usually one file per process thread [21, 22]. This simplifies their I/O behavior as each independent compute thread is able to independently output data with no job-wide coordination, and the I/O throughput of all of the I/O nodes associated with that job is available. The downside to this approach is that metadata operations on large-scale compute systems tend to be costly [7] [53] and writing out results as thousands of small files leads to poor filesystem performance.

As an alternative to this file-per-process model, libraries such as PLFS [5], Parallel HDF5 [17], and Parallel netCDF [29] 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 this common case without requiring application modification — only the addition of a custom data staging script external to the application itself. 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.

In such a case the `RSA_PREPROCESS` step could be used to unpack this tar file before job execution, and the RSA scheduler would ensure that the `RSA_PREPROCESS` step completes before the end-user's job could start execution on the compute system. This restriction would be implemented in the startup script[18] that is responsible for preparing the I/O nodes.

[18] Appendix K — **job-start.sh**

CHAPTER 4
Proof-Of-Concept System

4.1 Target Full-scale Architecture

The RAMDISK Storage Architecture was developed specifically to meet RPI's requirements for a next-generation supercomputing facility [9, 15]. The full-scale design calls for a 4-Terabyte RSA cluster (consisting of 32 nodes, each with 128GB of 1333 MHz DDR3 DRAM) interconnected to the compute system with 40Gbps Infiniband. The proposed compute system is a 512-node IBM Blue Gene/Q model 100. It is a 100TFlop system with 8 I/O nodes, each compute node consisting of a 16-core PowerA2-based CPU with 16GB of DDR3 DRAM [14, 33].

The proof-of-concept implementation demonstrated here 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 only differences between them are the decreased DRAM capacity, 32GB per node versus a target of 128GB, and node count, 16 nodes in the proof-of-concept versus 32 in the target system.

There is one critical difference in the network architecture of the proof-of-concept system compared to the proposed full-scale system. The network in the prototype has a bottleneck at the single Gigabit Ethernet link connecting the RSA cluster to the Blue Gene/L's functional network that the I/O nodes are connected to, whereas in the proposed environment there would be a non-blocking 40Gbps Infiniband fabric connecting the Blue Gene/Q I/O nodes to the RSA nodes.

4.2 Proof-Of-Concept Hardware Implementation

To test the entire RSA system, from the scheduling and dynamic allocation of the RAMDISK Storage Accelerator through to file access on the compute nodes, a proof-of-concept system was assembled using systems on the RPI campus. The RAMDISK Storage Accelerator was implemented by borrowing 16 nodes of the Scientific Computation Research Center's *Hydra* cluster [46]. 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 version 2.8.2 to construct the RAMDISKs.

The compute system used for testing is the RPI *SUR Blue Gene/L* [47], 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 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.

4.3 RSA Scheduler Implementation

The *BASH* scripting language ties the disparate systems together and constructs the RSA Scheduler itself. BASH is readily available on the three main components of the proof-of-concept system — the SUR Blue Gene front-end-node,

I/O nodes, and the RSA nodes — and is therefore an ideal choice for linking them together. Additionally, the interpreted nature of shell scripting makes it easier to modify the code to fit different systems, especially as modifications would be required to make the system run on the targeted full-scale Blue Gene/Q architecture.

4.3.1 RSA Scheduler

The main body of the RSA Scheduler[19] is implemented as an as asynchronous state-machine-driven single-pass routine. This routine is then executed on a fixed schedule to poll for system changes and take the appropriate next steps for each job. State transitions are denoted by changes in the status of the compute job — the `job-state` — and changes in the state of the RSA nodes — the `rsa-state`. On each pass through the core scheduling logic, jobs are first organized by their `job-state` and then the `rsa-state` within each subsection.

External call-outs are made to handle interaction with other components. This includes creating and destroying RAMDISKs on demand, managing RAMDISK access on the Blue Gene I/O nodes, as well as starting and stopping data-staging scripts. These external programs will update the `rsa-state` upon completion. The RSA Scheduler will then be able to take the appropriate next step on a later pass through the scheduling logic. Figure 3.1 shows the different `job-state` transitions. Note that `job-state` transitions here are driven by the actions of the SLURM scheduler and the compute job, and not those of the RSA Scheduler itself and thus may occur at any time. Transitions between `rsa-state`s are handled based on the current `job-state`. Transitions between jobs in the `on-deck` state are given in Figure 3.2, between `demoted` jobs in Figure 3.3, `running` in Figure 3.4, and `finished` in Figure 3.5.

4.3.2 RSA Node Assignment

Eligible jobs in the `on-deck job-state` are assigned RSA nodes in a fixed proportion to the number of requested compute nodes by a separate function[20]. When sufficient nodes are available, they are assigned and removed from the list of

[19] Appendix A — **rsa-scheduler.sh**
[20] Appendix P — **try-to-assign-nodes.sh**

free nodes. Jobs that do not request RSA resources, as well as those requesting too few compute nodes[21] to be eligible to use the RSA, are moved to the `INELIGIBLE` `rsa-state`. These variables are determined by the script used to parse the user-submitted SLURM batch script[22].

4.3.3 RSA Construction and Destruction

Two call-out scripts manage creation[23] and destruction[24] of the RAMDISKs on the assigned RSA nodes. Construction is accomplished on each node by transferring the PVFS configuration over to the node, mounting the `/dev/ram0` RAMDISK on each node, and starting the PVFS server process. The newly created RAMDISK is mounted on the frontend node so that it is available to the data staging scripts. RSA destruction reverses this process by unmounting the RAMDISK from the control node, connecting out to stop the PVFS server processes and then unmounting the RAMDISK from each node. Call-outs to scripts installed on the RSA nodes handle the local setup[25] and destruction[26] processes on each node directly and can be tailored to support different RSA node configurations and parallel filesystems.

4.3.4 Data Stage-In and -Out

Separate scripts are used to stage data in[27] and out[28] of the RSA. Data staging in both directions is, by default, handled by the `rsync` command[29]. Data is copied from and to the job's RAMDISK and the directories set by the `RSA_DATA_IN` and `RSA_DATA_OUT` variables in the user-submitted job script. The default stage-in and stage-out scripts can be replaced with ones specified by the user through the

[21] Jobs requesting too few compute nodes would be proportionally allocated less than one RSA node. As the RSA nodes are meant to be dedicated to individual jobs such a request is ignored. An enhanced version of the RSA could provide fractional resources if desired by using smaller RAMDISKs on each RSA node and divvying them up between smaller-sized jobs.

[22] Appendix R — **parse-job-options.sh**

[23] Appendix D — **rsa-construct.sh**

[24] Appendix E — **rsa-deconstruct.sh**

[25] Appendix F — **create-ramdisk.sh**

[26] Appendix G — **destroy-ramdisk.sh**

[27] Appendix H — **rsa-stage-in.sh**

[28] Appendix J — **rsa-stage-out.sh**

[29] The `rsync` command recursively copies data from a given folder to a destination, while preserving file attributes such as access time and file permissions.

`RSA_STAGE_IN` and `RSA_STAGE_OUT` variables.

A separate script[30] cancels a running stage-in process in the event that a job has been demoted or if job execution has started before the data stage-in finished.

4.3.5 SLURM Integration

There are two points of integration within the SLURM scheduler itself. The first is a script launched immediately before the compute job begins execution as part of SLURM's *Prolog* script[31]. The second is launched by SLURM's *Epilog* routine[32] immediately after the compute process finishes.

The start script determines the current `rsa-state` and, if the RAMDISK is ready, mounts the RAMDISK on the I/O nodes (this is handled separately[33]). It then updates both the `rsa-state` and `job-state`.

The finish script unmounts the RAMDISK from the I/O nodes[34] and updates the `rsa-state` and `job-state`. The RSA Scheduler then starts the data stage-out process on its next pass.

4.3.6 Supplemental Scripts

A small monitoring utility[35] can be used to track the `rsa-` and `job-state` for active jobs. It is designed to run within the UNIX `watch` command. For example, running it as `watch -n 15 ./rsa-status.sh` will show updates of the status every 15-seconds.

Global environment variables affecting the RSA Scheduler installation are set centrally[36]. Tunable settings here include the `PROPORTION` of compute nodes to RSA nodes and the `DEFAULT_DELAY` between using the RAMDISK to stage data in and out.

Another utility[37] implements a function to translate between a SLURM job

[30] Appendix I — **rsa-stop-stage-in.sh**
[31] Appendix K — **job-start.sh**
[32] Appendix N — **job-finish.sh**
[33] Appendix L — **rsa-mount-on-ionodes.sh**
[34] Appendix M — **rsa-unmount-from-ionodes.sh**
[35] Appendix B — **rsa-status.sh**
[36] Appendix C — **global.sh**
[37] Appendix O — **job-to-ionodes.sh**

id and a list of the I/O nodes in the assigned Blue Gene block. An initialization utility[38] is given to initialize the `rsa-state` from scratch. Yet another utility[39] dynamically creates a PVFS configuration file corresponding to a given job's set of allocated RSA nodes, and can be modified to match the parallel filesystem used.

[38] Appendix S — **prepare-machine.sh**
[39] Appendix Q — **rsa-config-gen.sh**

CHAPTER 5
Results and Discussion

5.1 RSA Scheduler Results

A sample log file from running the RSA-scheduler on the SUR Blue Gene alongside eleven test jobs is attached as Appendix T. Jobs are shown running through most scheduling scenarios along with the timestamp for each event occurrence. This demonstrates that the RSA scheduler is functioning correctly, and is able to perform all required functions for enabling the RAMDISK Storage Accelerator architecture.

5.2 Prototype System Results

A series of test jobs were run to demonstrate performance improvements from using the RSA on the proof-of-concept system. A major 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. In the proposed full-scale system this bottleneck would not exist, as the RSA nodes would be directly connected to the same 40Gbps Infiniband fabric as the I/O nodes.

5.2.1 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 5.2.1. The test case was configured to write out a 2GB 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

Table 5.1: 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

filesystem performance) on a Blue Gene/L system is around 50MB/s [31, 56]. Thus, this result primarily demonstrates this same bottleneck.

5.2.2 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 5.2.2 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 [4, 19].

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

Table 5.2: 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

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 preferred. 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.

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 [6, 21, 22, 27, 38, 54], and this is clearly an instance where the RSA would be particularly beneficial.

5.2.3 Results for a Single File via MPI-IO over 1024 Nodes

A third test was then run to show results that are not directly influenced by contention in GPFS. By changing the write-out mode of the simulation to using a single shared file through MPI-IO, the same 2GB test dataset is produced while avoiding the metadata contention that caused GPFS problems in one-file-per-process mode. Results below substantial improvement for GPFS, although this is still 35% slower than writing results as file-per-process to the RAMDISK as seen in Section 5.2.2.

Table 5.3: Comparison of output performance of GPFS disk storage vs. RSA, 1024 nodes (2048 processes) using MPI-IO to write a single shared file. Times are in seconds.

	No RSA, MPI-IO
Run 1	43.595017
Run 2	51.371842
Run 3	54.810469
Mean	49.925776

Testing determined that MPI-IO is not supported on PVFS 2.8 filesystems with the Blue Gene/L MPI stacks[40]. Thus, results for the RSA for a single file via MPI-IO cannot be obtained on the proof-of-concept system. MPI-IO on PVFS2 will be supported by the target Blue Gene/Q architecture.

5.2.4 IOR Benchmark Results

The IOR synthetic filesystem benchmark [43] 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 5.2.4. Notably, the IOR benchmark times the initial delay in opening each file. This value directly demonstrates the overhead in GPFS for file creation.

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

[40]PVFS2 needs direct support in MPI-IO [40]. MPI-IO support is enabled through the MPI toolchain on each system. On a Blue Gene/L this is a proprietary package with support for the Blue Gene/L interconnect hardware, and an updated version is not available.

Table 5.4: 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

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* [50] 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 [13], 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.

5.3 RAMDISK Performance Results

As the results obtained from the proof-of-concept system do not demonstrate the true performance benefits available from the RSA due to the Gigabit Ethernet bottleneck in the proof-of-concept system, a further test was run to determine the potential performance of a single RSA node.

The *bonnie++* [11] benchmark was run on a single Hydra node against a locally created RAMDISK as a baseline performance measurement. The results given are for an `ext2` formatted 30 Gigabyte RAMDISK, the same configuration used to build out the PVFS2 parallel RAMDISK. Table 5.3 shows the results, notably an average write speed of 867 MBytes/sec and read of 3.4 GBytes/sec. A notable difference here is the write versus re-write performance. The Linux memory page management is responsible for the lower performance in the write case as it adds additional delay when assigning memory pages to the RAMDISK. In the re-write case these pages are

Table 5.5: Consolidated Results from the Bonnie++ benchmark on a single RSA node.

Test	Performance
Write, MBytes/sec	867
Rewrite, MBytes/sec	1,163
Read, MBytes/sec	3,484
File creation per second	7470

already assigned and can be quickly overwritten. Future work could address these difference and pre-allocate the entire space to the RAMDISK ahead of operation.

An additional set of tests could determine the scalability of the RSA systems within the Hydra cluster. However, during implementation it was discovered that software incompatibilities between current versions of PVFS2 and the Linux kernel prevented using the Hydra nodes from connecting to the PVFS2 filesystem as clients. While the PVFS2 server runs in userspace without issue on these systems, the PVFS2 kernel client module does not support the 2.6.37 kernel used on Hydra. At present, any kernel that would support the PVFS2 client module would not be able to support the Infiniband adapters available in Hydra, so any attempt at scaling results would be bound by the performance of the Gigabit Ethernet switch.

CHAPTER 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.

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] Agerwala, T. Exascale computing: the challenges and opportunities in the next decade. *ACM SIG-PLAN Notices 45*, 5 (May 2010), 1–2.

[3] Ali, N., Devulapalli, A., Dalessandro, D., Wyckoff, P., and Sadayappan, P. Revisiting the Metadata Architecture of Parallel File Systems. In *SC'08, 3rd Petascale Data Storage Workshop* (Austin, TX, Nov. 2008), ACM/IEEE.

[4] Argonne Leadership Computing Facility Wiki. I/O Tuning. 2010. URL: https://wiki.alcf.anl.gov/index.php/I_O_Tuning [Date last accessed: Nov. 8, 2011].

[5] Bent, J., Gibson, G. A., Grider, G., McClelland, B., Nowoczynski, P., Nunez, J., Polte, M., and Wingate, M. PLFS: A checkpoint filesystem for parallel applications. In *SC'09 Conference Proceedings* (Portland, OR, USA, Nov. 2009), ACM/IEEE.

[6] Bent, J., and Grider, G. Usability at Los Alamos National Lab. In *U.S Department of Energy Best Practices Workshop on File Systems and Archives* (Sept. 2011).

[7] 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.

[8] Braam, P. The Lustre Storage Architecture. In *The Conference on High Speed Computing* (Salishan Lodge, Gleneden Beach, Oregon, Apr. 2006), LANL/LLNL/SNL, pp. 24–25.

[9] 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 [Date last accessed: Dec. 1, 2011].

[10] Caulfield, A. M., Coburn, J., Mollov, T., De, A., Akel, A., He, J., Jagatheesan, A., Gupta, R. K., Snavely, A., and Swanson, S. Understanding the Impact of Emerging Non-Volatile Memories on High-Performance, IO-Intensive Computing. In *SC'10 Conference Proceedings* (New Orleans, LA, USA, 2010), ACM/IEEE.

[11] Coker, R. Bonnie++ Benchmark. URL: http://www.coker.com.au/bonnie++/ [Date last accessed: Nov. 6, 2011].

[12] 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.

[13] Dykstra, P. Protocol Overhead. URL: http://sd.wareonearth.com/~phil/net/overhead/ [Date last accessed: Nov 6., 2011].

[14] Feldman, M. IBM Specs Out Blue Gene/Q Chip. URL: http://www.hpcwire.com/hpcwire/2011-08-22/ibm_specs_out_blue_gene_q_chip.html [Date last accessed: Nov. 1, 2011].

[15] Feldman, M. Rensselaer Orders Up Blue Gene/Q for Exascale and Data-Intensive Research. Oct. 2011. URL: http://www.hpcwire.com/hpcwire/2011-10-25/rensselaer_orders_up_blue_gene_q_for_exascale_and_data-intensive_research.html [Date last accessed: Dec 1., 2011].

[16] Fitzpatrick, B. Distributed caching with memcached. *Linux Journal 2004*, 124 (Aug. 2004), 5–5.

[17] 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.

[18] Freitas, R. F., and Wilcke, W. W. Storage-class memory: The next storage system technology. *IBM Journal of Research and Development 52*, 4-5 (2008), 439–448.

[19] 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.

[20] Frings, W., Wolf, F., and Petkov, V. Scalable massively parallel I/O to task-local files. In *SC'09 Conference Proceedings* (Portland, OR, USA, 2009), ACM/IEEE.

[21] 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.

[22] 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.

[23] He, J., Jagatheesan, A., Gupta, S., Bennett, J., and Snavely, A. DASH: a Recipe for a Flash-based Data Intensive Supercomputer. In *SC'10 Conference Proceedings* (New Orleans, LA, USA, 2010), ACM/IEEE.

[24] 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.

[25] Jette, M. Release Notes for SLURM Version 2.3. July 28 2011. URL: https://github.com/SchedMD/slurm/blob/slurm-2.3/RELEASE_NOTES [Date last accessed: Oct. 28, 2011].

[26] Jette, M., and Auble, D. SLURM - Multifactor Priority Plugin. URL: http://www.schedmd.com/slurmdocs/priority_multifactor.html [Date last accessed: Oct. 28, 2011].

[27] 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.

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

[29] 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.

[30] 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.

[31] Matthews, B., and Wickberg, T. Network and Filesystem Performance Experiments on a 1-Rack Blue Gene/L Supercomputer, May 2010.

[32] 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.

[33] Morgan, T. IBM uncloaks 20 petaflops BlueGene/Q super. URL: http://www.theregister.co.uk/2010/11/22/ibm_blue_gene_q_super/ [Date last accessed: Nov. 5, 2011].

[34] Norman, M. L., and Snavely, A. Accelerating data-intensive science with Gordon and Dash. In *Proceedings of the 2010 TeraGrid Conference* (2010), TG '10, ACM, pp. 14:1–14:7.

[35] 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.

[36] 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.

[37] 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.

[38] 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.

[39] Prabhakar, R., Srikantaiah, S., Patrick, C., and Kandemir, M. T. Dynamic storage cache allocation in multi-server architectures. In *SC'09 Conference Proceedings* (Portland, OR, USA, Nov. 2009), ACM/IEEE.

[40] PVFS2 Development Team. A Quick Start Guide to PVFS2. URL: http://www.pvfs.org/cvs/pvfs-2-8-branch-docs/doc//pvfs2-quickstart/pvfs2-quickstart.php#SECTION00090000000000000000 [Date last accessed: Nov. 6, 2011].

[41] 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.

[42] Sahni, O., Zhou, M., Shephard, M. S., and Jansen, K. E. Scalable implicit finite element solver for massively parallel processing with demonstration to 160K cores. In *SC'09 Conference Proceedings* (Portland, OR, USA, 2009), ACM/IEEE.

[43] Scalable I/O Project, LLNL. IOR Benchmark. URL: http://sourceforge.net/projects/ior-sio/ [Date last accessed: Nov. 6, 2011].

[44] Schaller, R. R. Moore's Law: Past, Present and Future. *Spectrum, IEEE 34*, 6 (1997), 52–59.

[45] Schmuck, F., and Haskin, R. GPFS: A Shared-Disk File System for Large Computing Clusters. In *Proceedings of the FAST '02 Conference on File and Storage Technologies (FAST-02)* (Berkeley, CA, Jan. 2002), USENIX Association, pp. 231–244.

[46] SCOREC Wiki. Hydra - ScorecWiki. URL: https://www.scorec.rpi.edu/wiki/Hydra [Date last accessed: Oct. 28, 2011].

[47] SCOREC Wiki. SUR Blue Gene - ScorecWiki. URL: https://www.scorec.rpi.edu/wiki/SUR_Blue_Gene [Date last accessed: Oct. 28, 2011].

[48] Smirni, E., Aydt, R. A., Chien, A. A., and Reed, D. A. I/O Requirements of Scientific Applications: An Evolutionary View. In *High Performance Mass Storage and Parallel I/O: Technologies and Applications*, H. Jin, T. Cortes, and R. Buyya, Eds. IEEE Computer Society Press and Wiley, New York, NY, USA, 2001, ch. 40, pp. 576–594.

[49] Stevens, R., and et all, A. W. Report on Architectures and Technology for Extreme Scale Computing. In *Scientific Grand Challenges Workshop on Architectures and Technology for Extreme Scale Computing* (2009), U.S. Dept. of Energy — Office of Science.

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

[51] 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.

[52] Weil, S. A., Brandt, S. A., Miller, E. L., Long, D. D. E., and Maltzahn, C. Ceph: A Scalable, High-Performance Distributed File System. In *OSDI* (2006), USENIX Association, pp. 307–320.

[53] 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.

[54] Xing, J., Xiong, J., Sun, N., and Ma, J. Adaptive and scalable metadata management to support a trillion files. In *SC'09 Conference Proceedings* (Portland, OR, USA, 2009), ACM/IEEE.

[55] Yoo, A., Jette, M., and Grondona, M. SLURM: Simple Linux Utility for Resource Management. In *Lecture Notes in Computer Science: Proceedings of Job Scheduling Strategies for Parallel Processing (JSSPP) 2003* (2002), Springer-Verlag, pp. 44–60.

[56] 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.

APPENDIX A
rsa-scheduler.sh

```
1  #!/bin/bash
2
3  source global.sh
4
5  # find jobs that have started running
6  JOBS_RUNNING=`squeue -h --start -t R -o "%i"`
7
8  # find all jobs we currently have state info for
9  JOBS_WITH_STATE=`find $STATE_DIR/job -mindepth 1 -maxdepth 1 -type
       d -printf "%f\n"`
10
11 # find jobs that are on-deck, i.e. waiting for Resources
12 JOBS_ON_DECK=`squeue -h -t PD -o "%i %R" | grep Resources | cut -f
       1 -d ' '`
13
14 # jobs in CF (configuring) state. these are effectively running now
       ,
15 # but will not set job state to running until finished booting the
       block for the job
16 JOBS_CONFIGURING=`squeue -h -t CF,R -o "%i"`
17
18 # jobs we have state info for can be:
19 # - ONDECK - still on deck
20 # - DEMOTED - cancelled or moved back to waiting on "(Priority)"
21 # - RUNNING - actively running
22 # - FINISHED - done execution, but RSA can still be staging out
23
24 for job in $JOBS_WITH_STATE; do
25   still_on_deck=0
26
27   JOB_DIR=$STATE_DIR/job/$job
28   job_state=`cat $JOB_DIR/job-state`
29
```

```
30  if [[ $job_state == "DEMOTED" ]]; then
31    # DEMOTED jobs are forced to stay demoted until cleanup
          finishes
32    # even if they would be otherwise ONDECK
33    JOBS_DEMOTED="$job $JOBS_DEMOTED"
34    continue
35  fi
36
37  if [[ $job_state == "FINISHED" ]]; then
38    JOBS_FINISHED="$job $JOBS_FINISHED"
39    # skip out of this loop cycle, on to the next job
40    continue;
41  fi
42
43  if [[ $job_state == "RUNNING" ]]; then
44    JOBS_RUNNING="$job $JOBS_RUNNING"
45    # skip out of this loop cycle, on to the next job
46    continue;
47  fi
48
49  # find jobs that were previously on deck, and are still on deck
50  for j in $JOBS_ON_DECK; do
51    if [[ $job -eq $j ]]; then
52      still_on_deck=1
53      continue
54    fi
55  done
56
57  # find jobs that were previously on deck, and are now configuring
58  # this avoids a race condition between block boot time and the
       delay for the rsa scheduler
59  for j in $JOBS_CONFIGURING; do
60    if [[ $job -eq $j ]]; then
61      still_on_deck=1
62      continue
63    fi
64  done
65
```

```
66    # if the job was previously on deck, but is not on deck now (and
          not running or finished), demote it
67    if [[ $still_on_deck -eq 0 ]]; then
68      JOBS_DEMOTED="$job $JOBS_DEMOTED"
69    fi
70  done
71
72  # cycle through demoted jobs
73  for job in $JOBS_DEMOTED; do
74    # this job has been removed from on deck, we need to release its
          resources
75    # our cleanup actions depend on the state of the RSA block
76
77    JOB_DIR=$STATE_DIR/job/$job
78
79    job_state=`cat $JOB_DIR/job-state`
80    rsa_state=`cat $JOB_DIR/rsa-state`
81
82    echo `date +%s` "job $job has been demoted"
83
84    case $rsa_state in
85    BOOTING)
86      # let boot script finish; hard to tear-down mid-bootup
87      ;;
88    TEARDOWN)
89      # wait for teardown to complete
90      ;;
91    BOOTED|READY|ABORTED)
92      # tear down booted block
93      echo "DEMOTED" > $JOB_DIR/job-state
94      echo "TEARDOWN" > $JOB_DIR/rsa-state
95      ./rsa-deconstruct.sh $job &
96      ;;
97    STAGING_IN)
98      # kill staging script, this will set rsa-state to ABORTED once
            done and we'll finish cleanup on the next pass
99      if [[ $job_state == "ONDECK" ]]; then
```

```
100       # if job_state is ondeck, then this is our first pass at
              stopping the data staging
101       # we want to avoid trying to kill it every scheduler pass
102       echo "DEMOTED" > $JOB_DIR/job-state
103       ./rsa-stop-stage-in.sh $job &
104     fi
105     ;;
106   DESTROYED|ASSIGNED)
107     # release allocated RSA nodes
108     cat $JOB_DIR/rsa-nodes >> $STATE_DIR/rsa-node/free
109
110     # remove job state files
111     echo `date +%s` "removing state files for job $job due to
            demotion/destruction"
112     rm -r $JOB_DIR
113     ;;
114   INELIGIBLE|INITIAL)
115     # remove job state files
116     echo `date +%s` "removing state files for job $job due to
            demotion/destruction"
117     rm -r $JOB_DIR
118     ;;
119   esac
120 done
121
122 # handle "on-deck" jobs - jobs that are not currently running, but
      are next in line
123 for job in $JOBS_ON_DECK; do
124
125   JOB_DIR=$STATE_DIR/job/$job
126
127   if [ ! -d $JOB_DIR ] ; then
128     # this is a newly promoted job, create initial state
129
130     mkdir $JOB_DIR
131     echo "INITIAL" > $JOB_DIR/rsa-state
132     echo "ONDECK" > $JOB_DIR/job-state
133   fi
```

```
134
135   rsa_state=‘cat $JOB_DIR/rsa-state‘
136   case $rsa_state in
137   INITIAL)
138     # if this fails, we’ll loop back around again
139     # if it works we’ll end up in BOOTING next round
140     ./try-to-assign-nodes.sh $job # block on this
141     ;;
142   ASSIGNED)
143     echo "BOOTING" > $JOB_DIR/rsa-state
144     ./rsa-construct.sh $job &
145     ;;
146   BOOTING)
147     # still booting, do nothing
148     ;;
149   BOOTED)
150     # finished booting, begin staging
151     echo ‘date +%s‘ "starting data staging script for job $job"
152     echo "STAGING_IN" > $JOB_DIR/rsa-state
153     ./rsa-stage-in.sh $job &
154     ;;
155   STAGING_IN)
156     # wait for data to finish staging, do nothing
157     ;;
158   READY)
159     # waiting on job execution, do nothing
160     ;;
161   INELIGIBLE)
162     # skip this job, we can’t provide any resources for it
163     ;;
164   esac
165 done
166
167 # handle running jobs
168 for job in $JOBS_RUNNING; do
169   JOB_DIR=$STATE_DIR/job/$job
170   rsa_state=’’
171   [ -e $JOB_DIR/rsa-state ] && rsa_state=‘cat $JOB_DIR/rsa-state‘
```

```
172
173   case $rsa_state in
174   INITIAL)
175     # if this fails, we'll loop back around again
176     # if it works we'll end up in ASSIGNED next round
177     ./try-to-assign-nodes.sh $job # block on this
178     ;;
179   ASSIGNED|DESTROYED)
180     # build up the RAMDISK
181     echo "BOOTING" > $JOB_DIR/rsa-state
182     ./rsa-construct.sh $job &
183     ;;
184   INELIGIBLE)
185     # job not using RSA, skip
186     ;;
187   INUSE_IN)
188     # if the job has been running for $DELAY, then go to ABORTED so
          we can use it to stage data out
189     JOB_DELAY=`cat $JOB_DIR/rsa-delay`
190     JOB_START=`cat $JOB_DIR/job-start`
191     NOW=`date +%s`
192     if [[ ! -f $JOB_DIR/rsa-data-out ]]; then
193       # if we aren't going to be used to stage data out, skip out
            of here and leave us in INUSE_IN
194       continue
195     fi
196     if [[ $NOW -gt $(( $JOB_START + $JOB_DELAY )) ]] ; then
197       echo `date +%s` "switching from staging in to out for job
            $job"
198       echo "UNMOUNT_NEEDED" > $JOB_DIR/rsa-state
199     fi
200     ;;
201   UNMOUNT_NEEDED)
202     # unmount from I/O nodes so we can reuse
203     echo "UNMOUNTING" > $JOB_DIR/rsa-state
204     ./rsa-unmount-from-ionodes.sh $job UNMOUNTED &
205     ;;
206   INUSE_OUT)
```

```
207     # RSA is in use to stage data out, nothing to do
208     ;;
209   STAGING_IN)
210     # destroy the RAMDISK so we can rebuild it to stage data out,
            jumps to ABORTED next
211     echo "STOPPING_STAGING_IN" > $JOB_DIR/rsa-state
212     ./rsa-stop-stage-in.sh &
213     ;;
214   STOPPING_STAGING_IN)
215     # wait for rsa-stop-stage-in.sh to complete, it will set state
            to ABORTED when done
216     ;;
217   ABORTED|UNMOUNTED)
218     # tear the block apart so we can reuse it
219     echo `date +%s` "tearing down and rebuilding for job $job"
220     echo "TEARDOWN" > $JOB_DIR/rsa-state
221     ./rsa-deconstruct.sh $job &
222     ;;
223   BOOTING|MOUNTING|TEARDOWN)
224     # still booting, do nothing
225     ;;
226   BOOTED)
227     # mount on nodes as stage_out
228     echo "MOUNTING" > $JOB_DIR/rsa-state
229     ./rsa-mount-on-ionodes.sh $job INUSE_OUT &
230     ;;
231   esac
232 done
233
234 # handle finished jobs
235 for job in $JOBS_FINISHED; do
236   JOB_DIR=$STATE_DIR/job/$job
237   rsa_state=''
238   [ -e $JOB_DIR/rsa-state ] && rsa_state=`cat $JOB_DIR/rsa-state`
239
240   case $rsa_state in
241   BOOTING|MOUNTING|TEARDOWN|UNMOUNTING)
```

```
242     # do nothing, wait for current action to finish before
            destroying
243     ;;
244   UNMOUNT_NEEDED)
245     # unmount from I/O nodes, fix a race condition here
246     echo "UNMOUNTING" > $JOB_DIR/rsa-state
247     ./rsa-unmount-from-ionodes.sh $job UNMOUNTED &
248     ;;
249   INUSE_OUT)
250     # RSA was in use for staging data out, we need to launch the
            stage-out script
251     echo "STAGING_OUT" > $JOB_DIR/rsa-state
252     ./rsa-stage-out.sh $job &
253     ;;
254   STAGING_IN)
255     # job completed rather quickly... stop the stage in
256     echo "STOPPING_STAGING_IN" > $JOB_DIR/rsa-state
257     ./rsa-stop-stage-in.sh $job &
258     ;;
259   STOPPING_STAGING_IN)
260     # wait for rsa-stop-stage-in.sh to complete, it will set state
            to ABORTED when done
261     ;;
262   STAGING_OUT)
263     # do nothing, data is being pushed back to disk storage
264     ;;
265   DONE_STAGING_OUT|READY|INUSE_IN|ABORTED|BOOTED|UNMOUNTED)
266     # destroy, as staging has finished or the RSA was not used for
            output, only input
267     echo "TEARDOWN" > $JOB_DIR/rsa-state
268     ./rsa-deconstruct.sh $job &
269     ;;
270   DESTROYED|ASSIGNED)
271     # release allocated RSA nodes
272     cat $JOB_DIR/rsa-nodes >> $STATE_DIR/rsa-node/free
273     mv $JOB_DIR/rsa-log $STATE_DIR/logs/$job-rsa-log
274     rm -r $JOB_DIR
275     ;;
```

```
276   INITIAL|INELIGIBLE)
277     # remove job state files
278     echo `date +%s` "removing state files for job $job due to job
            completion"
279
280     mv $JOB_DIR/rsa-log $STATE_DIR/logs/$job-rsa-log
281     rm -r $JOB_DIR
282     ;;
283   esac
284
285
286
287 done
```

APPENDIX B

rsa-status.sh

```
1  #!/bin/bash
2
3  # simple monitoring script to get the RSA status
4  # suggest running as: watch -n15 ./rsa-status.sh
5  source global.sh
6
7  printf "%7s%20s%20s\n" job job_state rsa_state
8
9  for job in `find $STATE_DIR/job -mindepth 1 -maxdepth 1 -type d -
       printf "%f\n"`; do
10   job_state=""
11   rsa_state=""
12
13   [ -f $STATE_DIR/job/$job/job-state ] && job_state=`cat $STATE_DIR
       /job/$job/job-state`
14   [ -f $STATE_DIR/job/$job/rsa-state ] && rsa_state=`cat $STATE_DIR
       /job/$job/rsa-state`
15
16   printf "%7s%20s%20s\n" $job $job_state $rsa_state
17 done
```

APPENDIX C
global.sh

```
1  #!/bin/bash
2
3  export STATE_DIR="/tmp/rsa-state"
4
5  export SCRIPT_DIR=/gpfs/gpfs0/home/wickbt/thesis-scripts
6
7  # set to 1 to simulate runs only
8  export SIMULATE=0
9  #export SIMULATE=1
10
11 # proportion of compute nodes to each RSA node, set this according
       to system scale:
12 export PROPORTION=128
13
14 # default delay for RSA to switch from staging in to preparing for
       staging out
15 export DEFAULT_DELAY=300
```

APPENDIX D

rsa-construct.sh

```
1  #!/bin/bash
2
3  source global.sh
4
5  job=$1
6
7  JOB_DIR=$STATE_DIR/job/$job
8  LOG=$JOB_DIR/rsa-log
9
10 echo `date +%s` "constructing RSA block for job $job" | tee -a $LOG
11
12 if [ $SIMULATE -eq 1 ]; then
13   # simulate delay in starting RSA block
14   sleep 10
15 else
16   # connect to nodes and construct RSA block
17
18   PVFSCONF=$JOB_DIR/pvfs-$job.conf
19
20   for i in `cat $JOB_DIR/rsa-nodes`; do scp -q $PVFSCONF $i:/tmp ;
        done
21
22   #PVFS
23   for i in `cat $JOB_DIR/rsa-nodes`; do ssh $i /cluster/rsa/create-
        ramdisk.sh $job ; done 2> /dev/null
24
25   # give the PVFS filesystem a few seconds to stabilize
26   sleep 10
27
28   mount -t pvfs2 tcp://$i:3334/ramdisk$job $JOB_DIR/ramdisk
29
30 fi
31
```

```
32 echo ‘date +%s‘ "done constructing RSA block for job $job" | tee -a
       $LOG
33
34 # update RSA state for this job
35
36 echo "BOOTED" > $JOB_DIR/rsa-state
```

APPENDIX E

rsa-deconstruct.sh

```
1  #!/bin/bash
2
3  source global.sh
4
5  job=$1
6
7  JOB_DIR=$STATE_DIR/job/$job
8  LOG=$JOB_DIR/rsa-log
9
10 echo `date +%s` "destroying RSA block for job $job" | tee -a $LOG
11
12 if [ $SIMULATE -eq 1 ]; then
13   # simulate delay in destroying RSA block
14   sleep 10
15 else
16   # connect to nodes and destroy RSA block
17
18   umount $JOB_DIR/ramdisk
19
20   for i in `cat $JOB_DIR/rsa-nodes`; do ssh $i /cluster/rsa/destroy
       -ramdisk.sh $job ; done
21 fi
22
23 echo `date +%s` "finished destroying RSA block for job $job" | tee
     -a $LOG
24
25 echo 'DESTROYED' > $JOB_DIR/rsa-state
```

APPENDIX F

create-ramdisk.sh

```
1  #!/bin/bash
2
3  # create ramdisk on local node
4
5  JOB=$1
6
7  mkfs.ext2 -q -m 0 /dev/ram0
8  mount /dev/ram0 /ramdisk
9
10 # initialize the PVFS filesystem structures on this node:
11 /users/wickbt/pvfs/sbin/pvfs2-server /tmp/pvfs-$JOB.conf -f
12 # run PVFS on this node
13 /users/wickbt/pvfs/sbin/pvfs2-server /tmp/pvfs-$JOB.conf
```

APPENDIX G
destroy-ramdisk.sh

```
1 #!/bin/bash
2
3 # destroy ramdisk on local node
4
5 pkill -9 pvfs2-server
6
7 umount /ramdisk
```

APPENDIX H
rsa-stage-in.sh

```
1  #!/bin/bash
2
3  source global.sh
4
5  job=$1
6
7  JOB_DIR=$STATE_DIR/job/$job
8  LOG=$JOB_DIR/rsa-log
9
10 if [ $SIMULATE -eq 1 ]; then
11   echo `date +%s` "simulating staging data for job $job" | tee -a
         $LOG
12
13   # simulate delay in staging data
14   sleep 10
15
16 else
17   if [ -f $JOB_DIR/rsa-data-in ]; then
18     echo `date +%s` "staging data for job $job" | tee -a $LOG
19     rsa_in=`cat $JOB_DIR/rsa-data-in`
20     user=`cat $JOB_DIR/user`
21
22     if [ -f $JOB_DIR/rsa-stage-in ]; then
23       # custom data stage in script
24       command=`cat $JOB_DIR/rsa-stage-in`
25     else
26       # default to rsync
27       command="rsync -av"
28     fi
29
30     # run the rsync as the user to prevent security issues
31     su - $user -c "$command $rsa_in/ $JOB_DIR/ramdisk" >> $LOG 2>&1
32
```

```
33      sync
34
35      echo `date +%s` "finished staging data in for job $job" | tee -
            a $LOG
36    else
37      echo `date +%s` "no rsa-data-in directory set for job $job, no
            stage-in occuring" | tee -a $LOG
38    fi
39  fi
40
41  # update RSA state for this job
42  echo "READY" > $JOB_DIR/rsa-state
```

APPENDIX I

rsa-stop-stage-in.sh

```
1  #!/bin/bash
2
3  source global.sh
4
5  job=$1
6
7  JOB_DIR=$STATE_DIR/job/$job
8  LOG=$JOB_DIR/rsa-log
9
10 echo `date +%s` "stop the data staging process for job $job" | tee
       -a $LOG
11
12 if [ $SIMULATE -eq 1 ]; then
13
14   # simulate delay in stopping the processes
15   sleep 1
16
17 else
18
19   pkill -f -9 "rsa-stage-in.sh $job"
20
21 fi
22
23 echo `date +%s` "stopped data staging process for job $job" | tee -
       a $LOG
24
25 echo "ABORTED" > $JOB_DIR/rsa_state
```

APPENDIX J

rsa-stage-out.sh

```
1  #!/bin/bash
2
3  source global.sh
4
5  job=$1
6
7  JOB_DIR=$STATE_DIR/job/$job
8  LOG=$JOB_DIR/rsa-log
9
10 if [ $SIMULATE -eq 1 ]; then
11   echo `date +%s` "simulating staging data out for job $job" | tee
         -a $LOG
12   # simulate delay in staging data out
13   sleep 30
14
15 else
16
17   if [ -f $JOB_DIR/rsa-data-out ]; then
18     echo `date +%s` "staging data out for job $job" | tee -a $LOG
19
20     rsa_out=`cat $JOB_DIR/rsa-data-out`
21     user=`cat $JOB_DIR/user`
22
23     if [ -f $JOB_DIR/rsa-stage-out ]; then
24       # custom data stage out script
25       command=`cat $JOB_DIR/rsa-stage-out`
26     else
27       # default to rsync
28       command="rsync -av"
29     fi
30
31     # run the rsync as the user to prevent security issues
```

```
32      su - $user -c "$command $JOB_DIR/ramdisk/ $rsa_out/" >> $LOG
            2>&1
33
34      echo `date +%s` "finished staging data out for job $job" | tee
            -a $LOG
35    else
36      echo "no rsa-data-out directory set for job $job, no stage-out
            occuring"
37    fi
38  fi
39
40  # update RSA state for this job
41  echo "DONE_STAGING_OUT" > $JOB_DIR/rsa-state
```

APPENDIX K
job-start.sh

```
1  #!/bin/bash
2
3  # we're called by SLURM's prolog, unlike the rest of the scripts
4  # so we need an explicit path here
5  source /gpfs/gpfs0/home/wickbt/thesis-scripts/global.sh
6  export PATH=/bgl/local/slurm/bin:$PATH
7
8  job=$1
9
10 JOB_DIR=$STATE_DIR/job/$job
11 LOG=$JOB_DIR/rsa-log
12
13 if [ -e $JOB_DIR ]; then
14   rsa_state=`cat $JOB_DIR/rsa-state`
15   case $rsa_state in
16   BOOTING|BOOTED|STAGING)
17     echo `date +%s` "starting job $job with RAMDISK not ready" >>
         $LOG
18     # RAMDISK not usable at this point, will attempt to use in
         stage-out instead
19     ;;
20   READY)
21     echo `date +%s` "starting job $job with RSA enabled" >> $LOG
22
23     cd $SCRIPT_DIR && ./rsa-mount-on-ionodes.sh $job INUSE_IN
24     ;;
25   esac
26 else
27   # job has no state info, must have started immediately
28   # create state info here instead, more state will be added by rsa
       -scheduler
29
30   mkdir $JOB_DIR
```

```
31    echo "INITIAL" > $JOB_DIR/rsa-state
32
33    echo `date +%s` "starting job $job with no previous state" >>
          $LOG
34
35  fi
36
37  date +%s > $JOB_DIR/job-start
38  echo "RUNNING" > $JOB_DIR/job-state
```

APPENDIX L

rsa-mount-on-ionodes.sh

```
1  #!/bin/bash
2
3  # connect out and mount the RAMDISK on the I/O nodes
4
5  source global.sh
6
7  job=$1
8  newstate=$2
9
10 JOB_DIR=$STATE_DIR/job/$job
11 LOG=$JOB_DIR/rsa-log
12
13 if [ $SIMULATE -eq 1 ]; then
14   # running in simulation mode, delay here a bit to match actual
       implementation
15   sleep 10
16 else
17   # mount the RAMDISK on all the BG/L I/O nodes
18   ionodes=`./job-to-ionodes.sh $job`
19   first_ramdisk_node=`head -n 1 $JOB_DIR/rsa-nodes`
20   for ionode in $ionodes; do
21     echo `date +%s` "mounting on $ionode for job $job" >> $LOG
22
23     ssh $ionode mount -t pvfs2 tcp://$first_ramdisk_node:3334/
         ramdisk$job /ramdisk >> $LOG 2>&1
24
25     # add in the bind mounts
26     if [[ -f $JOB_DIR/rsa-data-in && $newstate == "INUSE_IN" ]];
         then
27       rsa_in=`cat $JOB_DIR/rsa-data-in`
28       ssh $ionode mount -o bind /ramdisk $rsa_in >> $LOG 2>&1
29     elif [[ $newstate == "INUSE_IN" ]]; then
```

```
30        # we're booting the block right now, but weren't using it to
              stage data in
31        # so jump ahead to using it to stage out
32        newstate="INUSE_OUT"
33      fi
34
35      if [[ -f $JOB_DIR/rsa-data-out && $newstate == "INUSE_OUT" ]];
            then
36        rsa_out=`cat $JOB_DIR/rsa-data-out`
37        ssh $ionode mount -o bind /ramdisk $rsa_out >> $LOG 2>&1
38      fi
39    done
40  fi
41
42  jobstate=`cat $JOB_DIR/job-state`
43  if [[ $jobstate == "FINISHED" ]]; then
44    # we've lost a race here, jump to UNMOUNT in the FINISHED job-
          state to clean up correctly
45    newstate="UNMOUNT_NEEDED"
46  fi
47
48  echo $newstate > $JOB_DIR/rsa-state
```

APPENDIX M

rsa-unmount-from-ionodes.sh

```
1  #!/bin/bash
2
3  # unmount RAMDISK from a job's I/O nodes
4
5  job=$1
6  rsastate=$2
7
8  JOB_DIR=$STATE_DIR/job/$job
9  LOG=$JOB_DIR/rsa-log
10
11 echo `date +%s` "umounting RSA from I/O nodes on job $job" >> $LOG
12
13 for ionode in `$SCRIPT_DIR/job-to-ionodes.sh $job`; do
14   # undo the bind mounts if they still exist
15   if [ -f $JOB_DIR/rsa-data-in ]; then
16     rsa_in=`cat $JOB_DIR/rsa-data-in`
17     ssh $ionode umount $rsa_in
18   fi
19
20   if [ -f $JOB_DIR/rsa-data-out ]; then
21     rsa_out=`cat $JOB_DIR/rsa-data-out`
22     ssh $ionode umount $rsa_out
23   fi
24
25   ssh $ionode umount /ramdisk
26 done
27
28 echo $rsastate > $JOB_DIR/rsa-state
```

APPENDIX N
job-finish.sh

```
1  #!/bin/bash
2
3  # we're called by SLURM's epilog, unlike the rsa-* scripts
4  # so we need an explicit path here
5  source /gpfs/gpfs0/home/wickbt/thesis-scripts/global.sh
6  export PATH=/bgl/local/slurm/bin:$PATH
7
8  job=$1
9  JOB_DIR=$STATE_DIR/job/$job
10 LOG=$JOB_DIR/rsa-log
11
12 echo `date +%s` "job $job finished, umounting RSA from I/O nodes"
       >> $LOG
13
14 # fix state transitions in FINISHED state to avoid a race condition
15 # don't try to unmount INELIBIBLE jobs, they never had mounts
       active
16 # if we weren't using it to stage data out, we set UNMOUNTED to
       quickly tear down
17 # otherwise, make sure we end up in INUSE_OUT in FINISHED to start
       stage-out
18 rsastate=`cat $JOB_DIR/rsa-state`
19 if [ $rsastate == "INELIGIBLE" ]; then echo "FINISHED" > $JOB_DIR/
       job-state; exit ; fi
20 if [ $rsastate != "INUSE_OUT" ]; then rsastate="UNMOUNTED"; fi
21
22 # note: we still attempt to unmount the filesystems, even if they
       were not mounted anyways
23 # (no side-effects from attempting if they aren't)
24 $SCRIPT_DIR/rsa-unmount-from-ionodes.sh $job $rsastate
25 echo "FINISHED" > $JOB_DIR/job-state
```

APPENDIX O
job-to-ionodes.sh

```
#!/bin/bash

# convert a SLURM job id to a list of associated ionodes for the
    block

job=$1

block=`scontrol show job $job|grep Block_ID|cut -f 2 -d =`
midplanes=`scontrol show block $block|grep MidPlanes|cut -f 2 -d =
    |cut -f 1 -d ' '`

case $midplanes in
"bp[000x001]")
  # full system
  seq 0 31
  ;;
"bp000")
  # one midplane
  seq 0 15
  ;;
"bp001")
  # one midplane
  seq 16 31
  ;;
"bp000"*)
  # some fractional part of the first midplane

  # translate bp000[0-16] into 0-16
  # or bp000[0] to 0
  sub=`echo $midplanes|sed "s/bp000\[//"|sed "s/\]//"`

  # note: end may be null, this is okay
  start=`echo $sub|cut -f 1 -d -`
```

```
32    end=`echo $sub|cut -f 2 -d -`
33
34    seq $start $end
35    ;;
36  esac | awk 'BEGIN{FS=" "}{print "ionode" $1 * 2}'
37
38  # the awk statement above doubles the node names;
39  # the SUR BG/L only has every-other ionode active
40  # so the bp000 block has ionode0,2,4,...,28,30
41  # (and no odd-numbered ionodes are active)
```

APPENDIX P

try-to-assign-nodes.sh

```
1  #!/bin/bash
2
3  # try to assign RSA nodes to a job
4  job=$1
5
6  source global.sh
7
8  JOB_DIR=$STATE_DIR/job/$job
9
10 # if job_nodes < proportion, then we can't assign an entire RSA
       node, so skip
11 JOB_NODES=`squeue -h -j $job -o "%D"`
12 if [[ $JOB_NODES -lt $PROPORTION ]]; then
13   echo `date +%s` "job $job too small, marking ineligible"
14   echo "INELIGIBLE" > $JOB_DIR/rsa-state
15   exit
16 fi
17
18 # look for RSA_DATA_IN and _OUT settings
19 ./parse-job-options.sh $job
20
21 # if DATA_IN and DATA_OUT are not set, set INELIGIBLE
22 JOB_STATE=`cat $JOB_DIR/job-state`
23 if [[ ( ( ! -f $JOB_DIR/rsa-data-in ) && ( ! -f $JOB_DIR/rsa-data-
       out ) )
24   || ( ( ! -f $JOB_DIR/rsa-data-out ) && $JOB_STATE == "RUNNING" )
         ]]; then
25   echo `date +%s` "job $job does not have RSA variables set,
         marking ineligible"
26   echo "INELIGIBLE" > $JOB_DIR/rsa-state
27   exit
28 fi
29
```

```
30 # ensure there are sufficient free RSA nodes for this job; if not
       skip over and try again later
31 RSA_NODES=‘echo $(( $JOB_NODES / $PROPORTION ))‘
32 FREE_NODES=‘wc -l $STATE_DIR/rsa-node/free|cut -f 1 -d ’ ’‘
33 if [[ $FREE_NODES -lt $RSA_NODES ]] ; then
34   echo ‘date +%s‘ "insufficient free RSA nodes for job $job, have
         $FREE_NODES need $RSA_NODES"
35   exit
36 fi
37
38 mkdir $JOB_DIR/ramdisk
39
40 head -n $RSA_NODES $STATE_DIR/rsa-node/free > $JOB_DIR/rsa-nodes
41 tail -n +$(($RSA_NODES+1)) /tmp/rsa-state/rsa-node/free > /tmp/rsa-
       state/rsa-node/free2
42 mv /tmp/rsa-state/rsa-node/free2 /tmp/rsa-state/rsa-node/free
43
44 echo ‘date +%s‘ "assigning $RSA_NODES rsa-nodes to job $job"
45
46 ./rsa-config-gen.sh $job
47
48 echo "ASSIGNED" > $JOB_DIR/rsa-state
```

APPENDIX Q
rsa-config-gen.sh

```
1  #!/bin/bash
2
3  . global.sh
4
5  job=$1
6
7  # generate a fresh configuration file for the ramdisk
8
9  PVFSGENCONFIG=/gpfs/gpfs0/home/wickbt/pvfs-for-levi/bin/pvfs2-
       genconfig
10
11 rsa_nodes=`cat $STATE_DIR/job/$job/rsa-nodes | awk 'BEGIN{ORS=","}{
       print $1}'`
12
13 PVFSCONF=$STATE_DIR/job/$job/pvfs-$job.conf
14
15 $PVFSGENCONFIG --quiet --protocol tcp --fsname ramdisk$job --
       ioservers $rsa_nodes --metaservers $rsa_nodes --storage /ramdisk
        $PVFSCONF
```

APPENDIX R
parse-job-options.sh

```
1  #!/bin/bash
2
3  # parse job script file for RSA_DATA_IN and _OUT options
4  # if set, verify directories exist and set appropriate job
       variables
5
6  source global.sh
7
8  job=$1
9
10 JOB_DIR=$STATE_DIR/job/$job
11
12 JOB_SCRIPT=`scontrol show job $job | grep Command | cut -f 2- -d =`
13
14 if [ -e "$JOB_SCRIPT" ]; then
15   data_in=`grep ^#RSA_DATA_IN $JOB_SCRIPT|cut -f 2- -d =`
16   data_out=`grep ^#RSA_DATA_OUT $JOB_SCRIPT|cut -f 2- -d =`
17   stage_in=`grep ^#RSA_STAGE_IN $JOB_SCRIPT|cut -f 2- -d =`
18   stage_out=`grep ^#RSA_STAGE_OUT $JOB_SCRIPT|cut -f 2- -d =`
19   delay=`grep ^#RSA_DELAY $JOB_SCRIPT|cut -f 2 -d =`
20   user=`scontrol show job $job|grep UserId|awk '{print $1}'|cut -f
         2 -d =|cut -f 1 -d '('`
21
22   echo $user > $JOB_DIR/user
23
24   if [[ -n "$data_in" && -d "$data_in" ]]; then
25     echo $data_in > $JOB_DIR/rsa-data-in
26   fi
27
28   if [[ -n "$data_out" && -d "$data_out" ]]; then
29     echo $data_out > $JOB_DIR/rsa-data-out
30   fi
31
```

```
32    if [[ -n "$stage_in" && -x "$stage_in" ]]; then
33      echo $stage_in > $JOB_DIR/rsa-stage-in
34    fi
35
36    if [[ -n "$stage_out" && -x "$stage_out" ]]; then
37      echo $stage_out > $JOB_DIR/rsa-stage-out
38    fi
39
40    if [[ -n "$delay" ]]; then
41      echo $delay > $JOB_DIR/rsa-delay
42    else
43      echo $DEFAULT_DELAY > $JOB_DIR/rsa-delay
44    fi
45  fi
```

APPENDIX S

prepare-machine.sh

```
1  #!/bin/bash
2
3  source global.sh
4
5  if [ -z "$STATE_DIR" ]; then
6    echo "no STATE_DIR set, aborting"
7    exit -1
8  fi
9  rm -rf $STATE_DIR
10
11 mkdir -p $STATE_DIR/job
12 mkdir -p $STATE_DIR/logs
13 mkdir -p $STATE_DIR/rsa-node
14
15 rsa_nodes="node[1-16]"
16 scontrol show hostnames $rsa_nodes > $STATE_DIR/rsa-node/free
```

APPENDIX T
Example RSA Scheduler Log

Example RSA Scheduler log file, showing a series of jobs running through the compute system. Timestamps are in seconds (Unix epoch time).

```
1320009608 job 1679 does not have RSA variables set, marking ineligible
1320009759 assigning 1 rsa-nodes to job 1716
1320009759 constructing RSA block for job 1716
1320009771 done constructing RSA block for job 1716
1320010272 staging data out for job 1716
1320010273 finished staging data out for job 1716
1320010287 destroying RSA block for job 1716
1320010289 finished destroying RSA block for job 1716
1320010378 assigning 1 rsa-nodes to job 1717
1320010378 constructing RSA block for job 1717
1320010390 done constructing RSA block for job 1717
1320010499 assigning 1 rsa-nodes to job 1718
1320010499 constructing RSA block for job 1718
1320010511 done constructing RSA block for job 1718
1320010514 staging data out for job 1717
1320010515 finished staging data out for job 1717
1320010529 destroying RSA block for job 1717
1320010531 finished destroying RSA block for job 1717
1320010590 job 1719 does not have RSA variables set, marking ineligible
1320010620 job 1719 has been demoted
1320010620 removing state files for job 1719 due to demotion/destruction
1320010650 staging data out for job 1718
1320010651 finished staging data out for job 1718
1320010665 destroying RSA block for job 1718
1320010667 finished destroying RSA block for job 1718
1320010711 job 1722 does not have RSA variables set, marking ineligible
```

```
1320010726 job 1722 has been demoted
1320010726 removing state files for job 1722 due to demotion/
   destruction
1320011480 assigning 8 rsa-nodes to job 1725
1320011481 constructing RSA block for job 1725
1320011503 done constructing RSA block for job 1725
1320011632 staging data out for job 1725
1320011845 finished staging data out for job 1725
1320011859 destroying RSA block for job 1725
1320011869 finished destroying RSA block for job 1725
1320012059 assigning 8 rsa-nodes to job 1726
1320012059 constructing RSA block for job 1726
1320012081 done constructing RSA block for job 1726
1320012210 staging data out for job 1726
1320012389 finished staging data out for job 1726
1320012391 destroying RSA block for job 1726
1320012401 finished destroying RSA block for job 1726
1320012573 assigning 8 rsa-nodes to job 1727
1320012573 constructing RSA block for job 1727
1320012595 done constructing RSA block for job 1727
1320012724 staging data out for job 1727
1320012902 finished staging data out for job 1727
1320012905 destroying RSA block for job 1727
1320012914 finished destroying RSA block for job 1727
```

APPENDIX U
Example SLURM Job Script for RSA

An example SLURM job file used to launch the "mpi-matrix" code used for testing. This specific version is shown making use of the RSA with custom stage-in and stage-out scripts.

```
#!/bin/bash

# set stage-in data directory:
#RSA_DATA_IN=/gpfs/gpfs0/home/wickbt/test8/input
# set custom stage-in script:
#RSA_STAGE_IN=/gpfs/gpfs0/home/wickbt/stage-in.sh

# set stage-out directory:
#RSA_DATA_OUT=/gpfs/gpfs0/home/wickbt/test8/results
# custom stage-out script
#RSA_STAGE_OUT=/gpfs/gpfs0/home/wickbt/stage-out.sh

# run in virtualnode mode
# matrix is 16384 elements x 16384 elements - 2GB in size
BGLMPI_MAPPING=TXYZ mpirun -mode VN -cwd `pwd` ./mpi-matrix 16384
```

Example custom stage-in script.

```
#!/bin/bash
# $1 is the RAMDISK directory, $2 is RSA_DATA_IN directory

# un-tar the data
tar xvf $2/input.tar $1
```

Example custom stage-out script.

```
#!/bin/bash
# $1 is the RAMDISK directory, $2 is RSA_DATA_OUT directory

# tar up the data
tar cvf $2/result.tar $1
```