Balancing Costs for Data Resilience

Zhao Zhang[*] Daniel S. Katz[+] Haoyuan Li[*] Kyle Chard[+]
Ian Foster[+] Michael J. Franklin[*] Ion Stoica[*]

[*]AMPLab, University of California, Berkeley
[+]Computation Institute, University of Chicago & Argonne National Laboratory

ABSTRACT

Resilience is critical for many-task applications running on large scale computing platforms in which node failures are inevitable. This is especially the case when using a transient in-memory file system for parallel scripting. The state of these scripts is stored, in practice, by the files they write, and these files are inherently volatile as they are not persisted to disk. Existing systems overcome node failure by providing resilient data access either through task re-execution or file replication. However, applying either technique blindly can be inefficient due to excessive recovery or replication costs associated with the irregularity of task runtimes and I/O. We present an adaptive approach that calculates and balances backup and recovery costs to guide resilience decisions. Our approach aims to identify at run time the best choices for a given computer system and application. We implement this adaptive approach inside the AMFORA parallel scripting framework and evaluate its effectiveness with an image processing application, Montage. We find that the adaptive approach recovers up to 57% faster than do pure re-execution or replication, while introducing only 2.9% overhead in failure-free performance on 64 compute nodes. We also discover that, at least for this application, backup decisions are relatively insensitive to node failure rate.

1. INTRODUCTION

***Zhao: the overall tone of this paper is about unreliable distributed computing platforms, I will make it more supercomputer, and declare we are emulating a supercomputer with Amazon EC2. Distributed computing applications have long strived to be resilient to unreliable computing nodes. As applications scale to thousands and tens of thousands of compute nodes, the likelihood of a single node failing during execution increases proportionally [17, 13]. ***Zhao: below is new text to shape the platform as a supercomputer, and declare what we are doing, and the goals We seek to build a resilient parallel scripting framework to enable many-task applications on computing platforms such as supercomputers. We model such platforms as an execution environment comprised of multiple nodes, each of which can independently store data. Nodes are unreliable; if a node fails, the data that it stores is lost. Our resilience mechanisms aim to enable continued (and uninterrupted) application execution in the presence of node failures. Importantly, we aim to minimize overhead in the failure-free case while providing rapid recovery in the presence of a failure.

***Kyle: Could delete the following paragraph? In parallel computing, much previous research has focused on checkpointing process states to stable storage in order to tolerate failures. Researchers have investigated, for example, the optimum checkpointing interval [48, 11], coordinated checkpointing [10], and consistent checkpointing [14]. These works all view the computation as a long-running parallel task and use a fixed checkpointing interval throughout. Upon failure, the processes halt execution, reach a consensus, roll back to the checkpoint, and then resume execution. Common optimization goals in these approaches are minimizing: the time lost due to failures, failure-free running time (end-to-end time to solution of an application execution without any failure), and coordination overhead (e.g. minimizing the number of messages for all processes to reach a consistent checkpoint).

In the MapReduce [12] and Many Task Computing [38] models, the abstraction of a single, long-running task is replaced by many, often short-lived tasks that are linked by producer-consumer data sharing relationships. Many scientific applications are also naturally composed of numerous small tasks [39]. This short task abstraction removes the need for coordinated checkpointing: we can think of tasks of atomic units that either execute to completion or fail, and rethink system resilience in terms of ensuring the availability of the data that flows between tasks. It then becomes possible to recover from node failure simply by recreating data located on the failed node. If we have preserved either metadata describing how to re-execute a task (a *lineage* backup) or the data itself (a *replication* backup), we can achieve this by 1) re-executing the task(s) that produced the missing data or 2) accessing copies of that data located on other nodes, respectively. Meanwhile, all other tasks can continue running as they are not affected by the failure.

The two backup and recovery approaches described above have been employed in various systems. For example, the Spark [49] and BAD-FS [5] systems re-execute tasks following node failure, while RAID 1 [33] and the Hadoop Distributed File System (HDFS) [6] replicate each file (at the block level) a configurable number of times to preserve data access following node failure. However, irregular execution times, data sizes, and data dependencies can cause problems for each approach when applied to many-task applications. Universal application of the lineage technique can result in excessive recovery costs, since a failure can require restarting deep in the lineage graph. On the other hand, ubiquitous application of replication may result in excessive overheads when failures are rare, due to data copying and storage costs. Thus, we present here a new configurable adaptive model that dynamically chooses between these two approaches, based on online, model-driven estimates of backup and recovery costs. We integrate this adaptive approach into AMFORA [50], a parallel scripting framework for scientific computing, and evaluate its performance in the context of a large many task scientific application, Montage [27, 24]. Our experiments show that when emulating a supercomputer with 64 m3.large Amazon EC2 instances, the adaptive model recovers up to 57% faster than a purely re-execution or replication approach, and introduces only 2.9% of failure-free overhead.

Kyle: Could these assumptions come later? In this paper, we assume the network that connects nodes has a uniform bandwidth and latency to simplify the discussion; we later show that this works reasonably well in practice. In the case of node failure, we assume that the data stored in that node is not accessible, which is true in the context of a supercomputer where there is no local disk.Zhao: add more words to previous sentence to make it supercomputer We assume lineage information gathering and data replication are synchronous. To simplify the discussion, we also assume that all nodes share a common, consistent failure rate and failures are uncorrelated. However, we later show that the failure has little impact on the adaptive backup model in Section 5.4.

The contribution of this work is the use of a general adaptive mathematical model for making dynamic backup***Kyle: is backup the right word here? maybe recovery, resilience or replication? decisions and our evaluation of this model in a practical setting. The model is applicable to a broad range of applications running on clouds, clusters, and supercomputers. We also show that node failure rate is irrelevant to backup choice decision in practice.

The rest of paper is organized as follows: Section 2 briefly introduces Montage and AMFORA, including AMFORA's architecture, programming model and data management. Section 3 introduces the task abstraction, the analytical model, and the dynamic replication algorithm. Section 4 describes the changes made to AMFORA to implement the resilience model, including solutions to additional technical problems. Section 5 presents and analyzes the performance of the resilience model. Section 6 surveys previous work in system resilience. Finally, Section 7 summarizes the work and envisions future research.

2. BACKGROUND

In this section, we briefly introduce the Montage application and AMFORA's programming, data management, and task management models.

2.1 Montage

Montage is an astronomy image processing application that assembles large mosaics from multiple small images obtained from telescopes, while preserving the amount and position of the energy. Figure 1 shows the data flow of montage application. Table 1 explains the application stage by stage.

Table 1: Montage tasks

Stage	Description
mProject	reads image files and writes reprojected images
mImgtbl	takes the one line from mProject output, and concats them into one file
mOverlaps	analyzes the image table, produces a metadata table describing which images overlap along with a task list of `mDiffFit` tasks (one for each pair of overlapping images)
mDiffFit	inputs two overlapping output files from `mProject`, fits a plane to the overlap region
mConcatFit	gathers all output data from the previous stage (coefficients of the planes), and summarizes them into one file
mBgModel	analyses the metadata from `mImgtbl` and the data from `mConcatFit`, creating a set of background rectification coefficients for each image, then generates a `mBackground` task list
mBackground	applies coefficients to the reprojected images
mAdd	reads output files from `mBackground`, and writes an aggregated mosaic file

2.2 AMFORA

AMFORA is a POSIX-compatible parallel scripting framework that allows users to run (unmodified script-based) programs in parallel with data stored in the distributed RAM-based AMFORA File System. AMFORA provides a simple programming interface to its task and data management capabilities. In essence, the programmer specifies many task computations in terms of operations performed on files, with operations on directories serving to specify collective transformations on many files.

2.3 AMFORA Programming Model

AMFORA script code for the first two stages of the Montage application is shown in Listing 1.

Listing 1: Parallel Script for Montage

```
#!/bin/bash

#mProjectPP
#for each file in rawdir/ we run a
#mProject task with it as input file
#and produces an output file stored
#in tempdir/

#Queue is the API to push a task into
#the queue
#Execute is the API to run the queued
#tasks in parallel

mkdir tempdir/
for file in 'ls rawdir/'
do
  Queue mProject rawdir/${file} \
    tempdir/hdu_${file}
done
Execute

#Gather API moves all files stored in
#tempdir/ to a single node
```

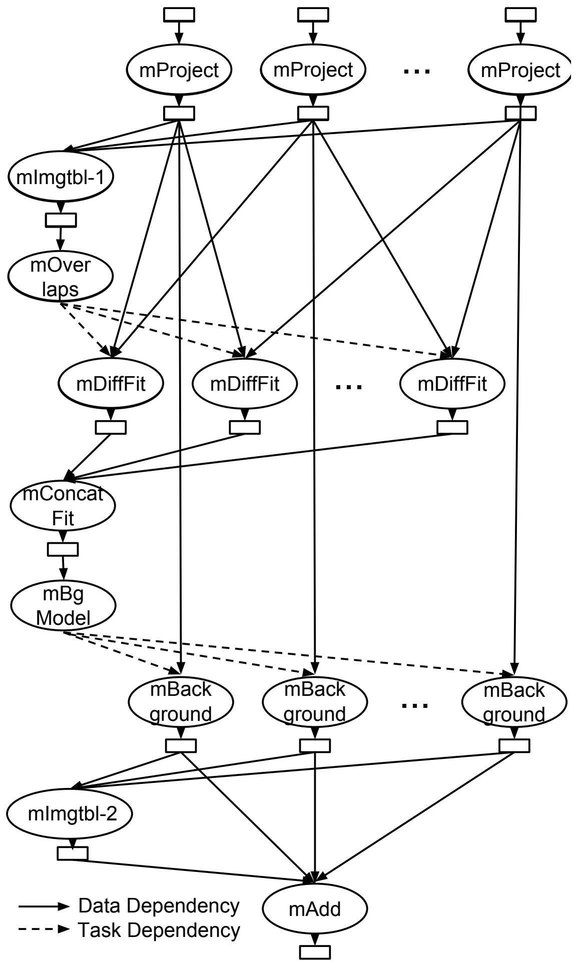

Figure 1: Montage data flow. Ovals represent tasks and boxes represent files. Solid lines show data dependencies, dashed lines show task dependencies.

```
24
25 #Then the mImgtbl task is launched to
26 #processes the files and write outputs
27 #to images.tbl
28 Gather  tempdir/
29 mImgtbl tempdir/ images.tbl
```

Lines 14–19 enqueue tasks; line 20 dispatches all queued tasks to available compute nodes for execution. Line 28 is a simple example of AMFORA collective data movement: it moves all files in the `tempdir/` directory from multiple nodes to a single node using a minimum-spanning tree algorithm. AMFORA supports multicast, gather, scatter, allgather, and shuffle (alltoall) data flows.

2.4 AMFORA Data Management

The AMFORA File System manages three types of data: directory metadata, file metadata, and file data. Every compute node in an AMFORA system is both a metadata server and an I/O server. The file system implements multi-reader single-writer consistency: a file can be read multiple times from multiple processes but can only be written by a single process once. Once a file is released (a POSIX primitive) from writing, it cannot be modified further.

All directory metadata is managed synchronously across all compute nodes. This synchronization does not represent a significant bottleneck as the creation and mutation of directories is rare in many-task applications [25]. File metadata is mapped across compute nodes via consistent hashing. The hash function assumes a ring topology composed of all compute nodes and, for each file, uses the hash of the file path to find the node that stores that file's metadata. File data is primarily stored where it is produced, so as to avoid remote writes. AMFORA does not support the partitioning or distribution of files across multiple nodes. While a limitation and a potential area for future work, this restriction does not represent a significant limitation in practice, as file sizes are generally small in many-task applications [25].

Table 2: Resiliency model terms.

Term	Meaning
P	rate of failure, i.e., $\frac{1}{MTTF}$
$timeout$	time required to switch from one replica of a data item to another, e.g., due to waiting for timeout
T	time to solution of a task without failures
r	number of metadata and file replicas
B	network bandwidth (assumed uniform)
$U_{repl}(x)$	time to backup data item x using replication
$U_{line}(x)$	time to backup data item x using lineage
$E_{repl}(x)$	time to recover data item x using replication
$E_{line}(x)$	time to recover data item x using lineage
x_i	the ith input file of a task
y	an output file of a task
$\|y\|$	size of file y
$\|M_y\|$	size of file y's metadata including the lineage information that can reproduce y

2.5 AMFORA Task Management

Tasks are launched by the AMFORA Execution Engine, which subsequently monitors their execution and collects runtime information that is subsequently used as input to the backup decision process. Information collected includes per-task queue time, start time, and end time, plus file usage (input or output) based on observations of runtime state mutations.

3. ADAPTIVE BACKUP MODEL

Our resilience approach builds on an analytical model that relates the costs of backup and recovery to system parameters such as communication cost and failure rate. Here we explain this model in detail and present the algorithm that we use to make dynamic replication decisions.

In general, we view a task as a function t that takes n input files and produces m output files:

$$y_1, \ldots, y_m = t(x_1, \ldots, x_n) \tag{1}$$

Table 2 defines these and other terms used in subsequent discussion.

3.1 Backup Cost

***Kyle: we seem to flip flop between calling it re-execution or lineage. I vote for re-execution and talk about that based on lineage metadata as you do here A file can be recovered either by retrieving a replica or through re-execution based on its lineage metadata. To create r replicas of a file or its lineage, $r-1$ additional copies are needed. The time required to synchronously create the $r-1$ replicas of a file y to memory is as follows:

$$U_{repl}(y) = \frac{|y|}{B}(r-1) \tag{2}$$

Similarly, the time required to synchronously replicate the lineage of a file y $r-1$ times is as follows.

$$U_{line}(y) = \frac{|M_y|}{B}(r-1) \tag{3}$$

3.2 Expected Recovery Cost

We can now estimate the expected recovery time via replication or re-execution, given the assumptions above.

3.2.1 Expected Replication Recovery Cost

If a file y is replicated on multiple nodes, then the expected replication recovery cost $E_{repl}(y)$ of file y can be evaluated as the sum of the expected recovery cost in the following cases:

The first replica is available: $(1-P)\frac{|y|}{B}$

The second replica is available but not the first:
$P(1-P)\left(\frac{|y|}{B} + timeout\right)$

etc.

where *timeout* is the time for the system to switch from one replica to another.

We add these items and transform the equation to this form:

$$E_{repl}(y) = \frac{|y|}{B}(1-P^r) + \left(\frac{P-P^r}{1-P} - (r-1)P^r\right) timeout \tag{4}$$

As P^r is small, we can replace it with 0 to obtain the following closed form approximation:

$$E_{repl}(y) \approx \frac{|y|}{B} + \frac{P}{1-P} timeout \tag{5}$$

***Kyle: this is the first time spatial appears - might be worth explaining or calling it spatial and temporal replication throughout. Equation 5 defines the replication recovery cost for an output file y to be approximately equal to the time required to transfer the file plus the time required to switch to the replica in the case of failure, where the likelihood of needing to use the replica is based on the rate of failure P Thus, when there is no failure, the cost of replication is simply the cost of creating the replicas

3.2.2 Expected Lineage Recovery Cost

The recovery time via re-execution is dependent on both the time to re-execute the task as well as the time to recover the required input files of that task. A task t_i has n input files. Assume that those input files are located on n other nodes. The probability that a node is available is $(1-P)$. The probability that all n nodes are available is $(1-P)^n$. The expected recovery time through lineage (i.e., re-execution) is $(1-P)^nT$. If there are one, two, or more failed nodes, the expected recovery time through lineage is:

One failure: $\binom{n}{1}P(1-P)^{n-1}(T+E(x_i))$

Two failures:
$\binom{n}{2}P^2(1-P)^{n-2}(T+E(x_i)+E(x_j))$

etc.

where $E(x_i)$ refers to the recovery cost of item x_i via the technique with which file x_i is backed up.

Adding these items gives the following closed form for the expected lineage recovery cost:

$$E_{line}(y) = T + P\sum_{i=1}^{n} E(x_i) \tag{6}$$

Equation 6 defines the lineage recovery cost for an output file y to be equal to the time to re-execute the task T plus the sum of the time required to recover all of T's input files. Here, the only overhead (in the case without failure) is related to the cost of updating file metadata with the information needed to re-execute the task.

3.2.3 Combining Backup and Recovery Cost

To simplify our discussion, we assume a task writes its output files first to local cache, then backs up the files synchronously with the specified technique. Thus the total time taken by a task is the sum of three parts: **T**: task execution time, including computation and I/O cost, **U**: time to backup the output file, and **E**: expected cost to recover the output file.

To take both the backup cost and expected recovery cost into account, we use a linear combination of **U** and **E**, which we call S, shown in Equation 7:

$$S = \alpha * U + (1-\alpha) * E \tag{7}$$

For every output file, we evaluate the total cost with both lineage or replication as the backup choice, and use the technique with the lower total cost. The term α in Equation 7 is a weight, which the user can set to between 0 and 1. If the user's optimization goal is backup cost, users can specify α close to 1 That means that execution will optimize the backup cost over recovery cost. In contrast, if the user's optimization goal is recovery cost, α should be specified close to 0. The system make backup decisions to optimize the expected recovery cost since the chance of failure is high. Section 5.3 quantitatively evaluates the impact of optimization weight on adaptive backup decision making.

3.2.4 Discussion

***Kyle: I'm not sure if this section adds a lot, I'm not sure if the discussion is based on what you think? or are we forward referencing results? I would vote to remove (although the picture is nice). We next discuss how task and system parameters impact the choice of backup approach. The four example tasks in Figure 2 vary in their computation time, I/O size, and number of input files. If we control the comparison by varying only one factor, we see some trends about backup decisions. An output file produced by a longer-running task will tend to be backed up by replication because the cost of regenerating the file by re-executing the task will be large. A larger output file is more likely to be backed up through lineage, since replicating this file takes a long time. A file that depends on more input files has a higher probability to be backed up through replication, as it takes a long time to read many input files during reproduction of this particular file.

We also observe the implication of system bandwidth for the choice of backup approach. Higher bandwidth improves replication performance more significantly than lineage; thus, other things being equal, the same file is more likely to be backed up through replication on a higher-bandwidth system.

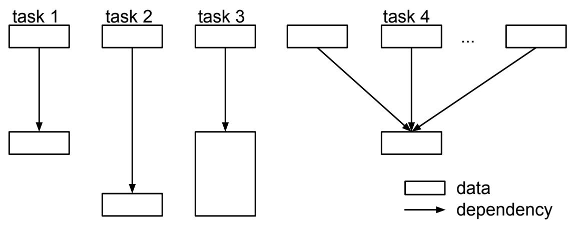

Figure 2: Example tasks in a three dimensional space of computation time, I/O size, and number of input files. Rectangles are data, rectangle area is proportional to data size, arrows are data flows, and the length of arrows indicates task computation time.

4. IMPLEMENTATION

To integrate the adaptive backup model into the AMFORA framework, we implemented two backup primitives: replicate() and lineage(). The *replicate()* primitive creates a configurable number of replicas for a file on peer nodes. The *lineage()* primitive records a file's lineage (i.e., the command line used to produce it) in the file's metadata. Also recorded in the file metadata are the file's host address and expected recovery cost***Kyle: Is this expected recovery cost for both methods? or just lineage? . The execution engine captures task runtime information such as time-to-solution and I/O size. We also require that it identify which parameters in the command line are input files and which are output files.

Both files and their associated metadata are copied to peer nodes a configurable number of times. File replicas are placed on nodes directly adjacent to the file hosting node in the ring topology. Similarly, metadata are copied to nodes adjacent to the metadata hosting node. Figure 3 shows an example data layout in which three copies each have been created for both the data (`data[f]`) and metadata (`meta[f]`) of a file `f`. To access file data or metadata, a client applies a consistent hash function, with the file path and node list as input, to determine the node that hosts the file data or metadata. If access to that node fails, it can switch to the next copy until the required item is returned. (The client can infer locations of copies beyond the first based on global topology information.

Recovery of a file for which no copy exists requires access to the file's lineage metadata. This metadata is retrieved and the computation that it describes is examined, to determine whether the input file(s) needed to create that file are present. If they are not, the process is repeated. The resulting directed acyclic graph of tasks is then executed.

Our implementation is also responsible for synthesizing lineage metadata and for maintaining the database of file sizes and task execution times that is used to guide backup decisions. To this end, the execution engine tracks modifications to all local metadata and data in order to gather task execution times and file usage patterns. This information allows each compute node to make individual backup decisions for each new file that is generated.

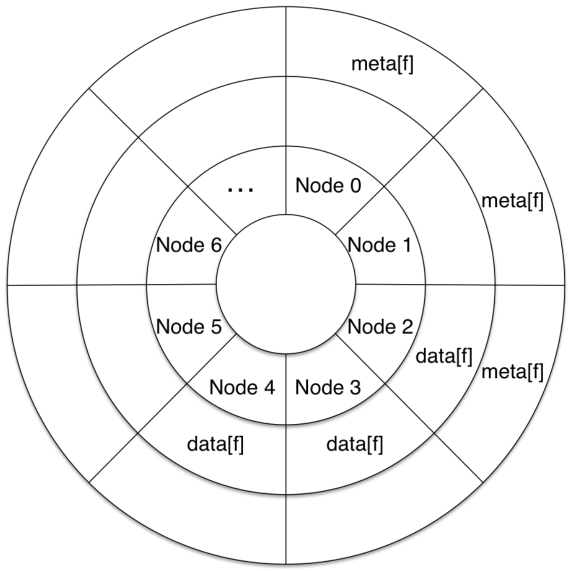

Figure 3: A data layout example with three-way replication. Node 2 produces file f, and Node 0 hosts f's metadata. Node 0 is chosen to host f's metadata by hashing f's file path. The actual data of f is replicated to Nodes 3 and 4, and the metadata is replicated to Nodes 1 and 2.

5. PERFORMANCE EVALUATION

We evaluate our approach by emulating a supercomputer environment with an Amazon EC2 cluster of 64 m3.large instances, each equipped with 2 Intel Xeon E5-2670 vCPUs and 7.5 GB memory. ***Zhao: rewrite the previous sentence to make it supercomputer

The Montage test case is a 3x3 degree image mosaic of 2MASS data centered at Galaxy m101. The application has 369 initial input files (raw images) and one final output file (mosaic). The application has nine stages (mImgtbl is executed twice). The mProject, mDiffFit, and mBackground stages have multiple tasks. The mImgtbl, mConcatFit, and mAdd stages have a single task but a large number of input files. Table 3 summarizes the basic statistics of each stage of the Montage test case.

The Montage application can also be viewed as eight independent applications that each have different computation and I/O distribution. The nine stages spans a wide space of many-task applications across dimensions of relative data/-computation intensiveness and parallelism (single task stage and distributed stage). ***Zhao: add previous paragraph to argue the coverage of Montage

In our recovery cost model we specify a network bandwidth (B) of 20 MB/s, which was measured in the EC2 cluster. The optimization weight (α) is set to 0.5, which means backup cost and expected recovery cost are treated equally. Since the overall application runs in 200 seconds, we set the rate of failure (P) as $\frac{1}{12800}$ (one fault per run).

5.1 Failure-free Backup Overhead

We first run a set of experiments to measure costs incurred by our resilience features in the absence of failure. We run the nine stages of the Montage application on our

Table 3: Number of tasks, inputs, and outputs, and input and output size, for each Montage stage

Stage	# Tasks	# Inputs	# Outputs	# Inputs per Task	# Outputs per Task	Max Input Size (MB)	Max Output Size(MB)	Input Dependency
mProject	369	369	738	1	2	2.1	4.2	filesystem
mImgtbl-1	1	369	1	369	1	4.2	0.97	mProject
mOverlaps	1	1	1	1	1	0.97	0.13	mImgtbl
mDiffFit	1065	369	2030	2	2	4.2	0.5	mProject
mConcatFit	1	1065	1	1065	1	0.5	0.5	mDiffFit
mBgModel	1	2	1	2	1	0.5	0.5	mImgtbl, mConcatFit
mBackground	369	369	369	1	1	4.2	4.2	mProject
mImgtbl-2	1	369	1	369	1	4.2	0.97	mBackground
mAdd	1	369	1	369	1	4.2	1200	mImgtbl-2, mBackground

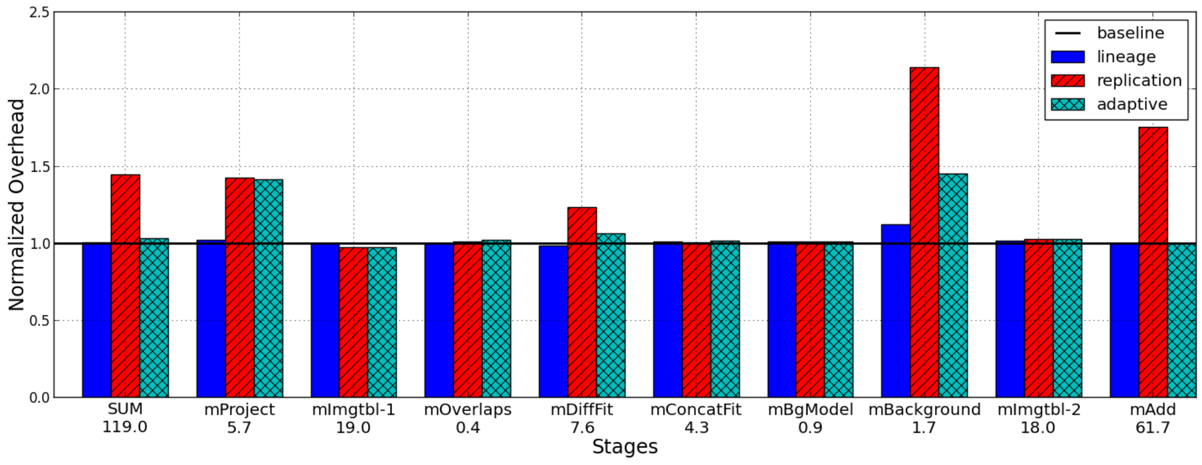

Figure 4: Montage performance comparison of lineage backup, replication backup, and adaptive backup resilience schemes without node failures during execution, shown as ratios between the time-to-solution of each resilience scheme against the no-resilience time-to-solution. Bars above the line indicates performance degradation, and bars below the line indicates performance improvements. The numbers underneath the stage labels are the baseline time-to-solution measured in seconds for overall application and each stage.

target platform using four different system configurations: no-resilience, lineage backup, replication backup (two replicas per file), and adaptive backup. Figure 4 shows, for each of the latter three cases, the resilience overhead expressed as the ratio between its execution time and the no-resilience time.

We see that in the no-failure case, the overhead incurred by the lineage approach is small (just 0.6%), as the only substantial additional work is that of inserting the generating task information into the file metadata. The replication scheme introduces a larger overall overhead of 44.0%, due to the need to replicate the output files to two other nodes.

At the individual Montage stage level, the lineage scheme incurs less overhead than the replication scheme in all stages except mImgtbl-1, which is just one task that reads all output files from mProject. With the replication scheme, mImgtbl-1 takes advantage of the locality of the replicas of the mProject output files.***Kyle: I dont understand what you mean by take advantage of replica locality? In the mBackground stages, the lineage scheme's overhead is significant (12.1%), because this stage involves many parallel, short tasks; the many concurrent metadata updates that result can introduce contention on the nodes that host the metadata. In contrast, the mProject, mDiffFit, mBackground, and mAdd stages all incur significant overhead when using replication, because they each produce many and/or large output files. For example, mDiffFit has over 2000 output files and mAdd has one task that writes one 1.2 GB output file.

The adaptive replication scheme incurs an overall overhead of 2.9%, which is inbetween the overheads from lineage and replication. For the mImgtbl-1, mOverlaps, mConcatFit, mBgModel, mImtbl-2 and mAdd stages, the adaptive scheme uses lineage backup. mDIffFit runs faster with the adaptive scheme than with replication, since its input files (outputs of mProjectPP) are backed up with replication. The mBackground stage runs in a time that is between that of the lineage and replication cases partially because the input files were replicated, and the output files are backed up through lineage.

Figure 5 shows the position of all output files in the test case on a two dimensional space of lineage and replication score***Kyle: cost? We never really define score. I dont mind calling it score but we should make sure that is clear earlier. calculated with Equation 7. Each point shows a single output file from each Montage stage. The points in the upper left triangle are those where the replication score is lower than the lineage store, while those in the lower right triangle are the files where the lineage score is lower than

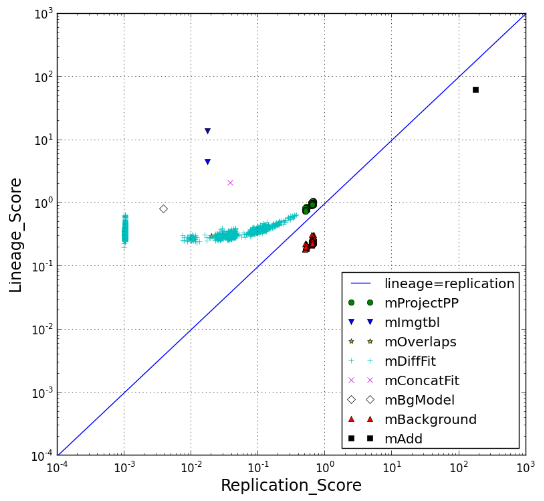

Figure 5: Montage output files' positioning in a two dimensional space of replication score and lineage score.

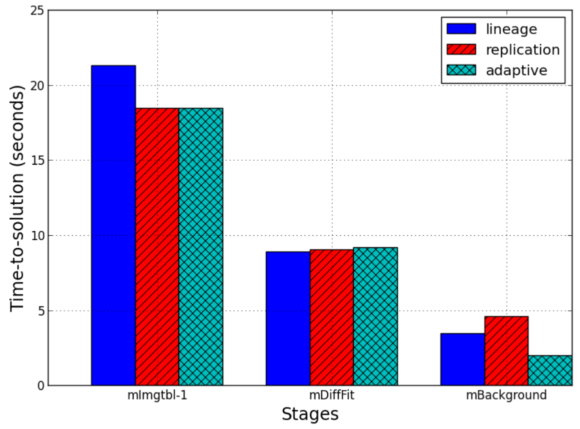

Figure 6: Recovery performance of mImgtbl, mDiffFit, and mBackground following a node failure that occurs after mProjectPP finishes and before mImgtbl starts.

the replication score.

5.2 Recovery Performance

To evaluate recovery time, we inject failures during job execution by causing a node to fail (fail-stop) at a specified application stage. We inject failures in two specific stages, namely at the end of the mProject stage and the end of the mBackground stage, in order to evaluate both single-stage recovery performance and multi-stage recovery (recursive recovery) performance.

5.2.1 *Single-Stage Recovery*

We first fail a node immediately after the mProject stage finishes, before the mImgtbl stage starts, and evaluate the impact on the mImgtbl-1, mDiffFit, and mBackground stages, each of which needs to access the output files of mProject. When one of these files is not available, the system must recover that file through either re-execution (based on lineage metadata) or by accessing a file replica (if the file replicated).

Figure 6 compares the times to solution of our three backup schemes for those three Montage stages. We see that the adaptive and replication schemes perform similarly in stages mImgtbl-1 and mDiffFit, as the output files of these two stages are backed up via replication; both also perform better than the lineage scheme. The mBackground stage runs faster under the adaptive scheme than in the lineage and replication scheme, as all of its tasks benefit from the previously replicated mProject output files and all of its output files are backed up through lineage as shown in Figure 5.

5.2.2 *Multi-Stage Recovery*

We fail a node immediately after mBackground finishes, before mImgtbl-2 starts. Both the mImgtbl-2 and mAdd stages need to access mBackground's output files. However, in this case, a node failure results in the loss of output files from not only mBackground but also mProject—files that we need in order to recover mBackground's outputs. Thus, upon file unavailability, the system recovers the mBackground output file in multiple stages (recursively), as follows: 1) access to one mBackground output file fails; 2) the system tries to recover that mBackground output file by re-executing the task; 3) that task accesses an output of mProject; 4) that mProject output is unavailable; 5) the system recovers the missing mProject output.

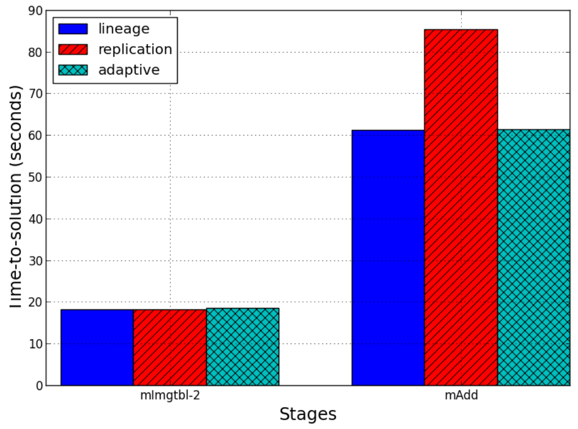

Figure 7: Recovery performance of mImgtbl and mAdd due to a node failure occurred right after mBackground finishes and before mImgtbl starts

Figure 7 shows the time-to-solution comparison between the three replication schemes. The adaptive scheme achieves similar performance with lineage and replication scheme for mImgtbl-2. This indicates that recovery from lineage recursively and backing up the mImgtbl-2 output file with the same scheme has identical performance to recovery using replication recursively and backing up mImgtbl-2 output files through replication. In fact, the adaptive approaches recovers the input files using lineage for mBackground and using replication for mProject. The mImgtbl-2 output file is backed up through replication. The adaptive scheme has significant performance advantages for the mAdd stage. It again achieves recursive recovery via lineage for mBackground and replication for mProject, but chooses to replicate the mAdd output file using lineage, which in turn results in better time-to-solution.

5.3 Impact of Optimization Weight

We use the output files of all stages of the Montage test case to evaluate how the optimization weight parameter α impacts the behavior of our adaptive algorithm***Kyle: scheme or algorithm? . We fix all parameters in the system except α, which we set variously to be 0.1, 0.5, and 0.9, indicating the user's concern with optimizing the failure-free run time.

Figure 8 shows our results. We see that as α increases, files move from the upper left triangle to the lower right triangle, with the result that the failure-free run time is shorter since more files are backed up through lineage. This result exactly matches the user's optimization goal of minimizing failure-free running time.

5.4 Impact of Failure Rate

The failure rate (P) of a computer system can be difficult to measure in practice. For example, Amazon EC2 reports system availability as a monthly uptime percentage [4]. The Blue Gene/Q supercomputer's MTTF is reported as 1–7 days [42], system-wide. Given a customized allocation size and hardware/software stack, it can be difficult for a user to convert these metrics to accurate MTTFs for a particular execution.

Surprisingly, the adaptive backup decisions remain the same under all five failure rate configurations***Kyle: what are these 5 configs? , which implies that the failure rate is irrelevant to the adaptive backup decision, at least for this application. The reason for this lack of sensitivity to P becomes clear when we look at Equations 5 and 6. We see that unless P is quite large, data and metadata copy costs dominate, and the lineage score and replication score have a simpler form than in Equation 5 and 6

$$S_{repl} = \alpha \frac{|y|}{B}(r-1) + (1-\alpha)\frac{|y|}{B}$$

$$S_{line} = \alpha \frac{|M_y|}{B}(r-1) + (1-\alpha)T$$

We also consider a second question, namely, how does the adaptive model respond to varying numbers of input files? For example, in the case of *task 1* and *task 4* in Figure 2, which are identical except for the number of input files, can the adaptive model***Kyle: We change between model, scheme, algorithm throughout determine that the output of *task 4* should be backed up through replication? (since more input files indicate a larger chance of failure) The answer is yes. Our computation of a task's time-to-solution T includes three parts: input access (either local or remote), computation time, and output to local RAM. If the computation and output phases take the same amount of time, more input files makes the input phase run longer, which is reflected in the measurement of T. The output of *task 4* thus has a higher recovery cost through lineage than that of *task 1*; thus, the adaptive scheme is more likely to replicate the data.

6. RELATED WORK

Fault tolerant resilience models have been explored in small-scale applications for single-threaded applications on individual PCs [29, 35], large-scale distributed workflows [45], and parallel and HPC computing applications [9, 43, 26], among others. In general, most resilience approaches rely on some form of data replication, either explicitly via replicating intermediate files or implicitly via checkpointing application state.

Replication of intermediate files is often the easiest approach for building resilient applications. Such algorithms can be employed trivially by an execution framework or file system without requiring application modification. Replication techniques are also used to enhance scalability and performance. While specific goals may differ, the replication techniques applied are similar. A brief summary of replication techniques in distributed storage environments is presented in a survey [1]. Replication techniques are often classified as either static or dynamic. In static configurations a set number of replicas and hosts are chosen at the start of the application lifecycle and are then used throughout execution. In a dynamic model these parameters are changed depending on access patterns, storage capacity, and bandwidth. Dynamic models often employ a decentralized architecture in which replicas are distributed over a peer-to-peer network [44] and decentralized decision mechanisms are used to place and access replicas. Replication techniques have been used for decades in distributed file systems such as the classic RAID [33] and more recently HDFS [6] and RAMCloud [32, 41] to replicate files over a distributed collection of nodes. ***Zhao: add one sentense The adaptive approach we present in this paper suits for a different environment where there are massive amount of compute node without local disks.

Lineage strategies have been explored in scientific workflows [40, 8, 16] as well as in distributed computing framework such as Spark [49]. Most often, lineage in scientific workflows is motivated by the need to meet deadlines and therefore focuses on concurrent task replication. Thus, tasks are actively replicated on several nodes concurrently and the first result is taken. For example, in the VGrADS project [40], tasks are selectively replicated based on resource reliability and application performance models. Similar techniques are used in volunteer computing projects [2] to establish consensus among several potentially untrusted sources. Spark, a data processing engine designed for processing Hadoop data, analyzes a task's *lineage graph*—the operations used to build it—in order to reexecute entire tasks without requiring replication. This approach is particularly advantageous in the Spark context as individual tasks are short (50-200 ms) stream-based computations. ***Zhao: add one sentense While the many-task applications running time and I/O amount can be highly irregular, so that no single approach can be the best solution.

Replication strategies have also been used to compute in the presence of faults, such as in algorithm-based fault tolerance (ABFT) [23], where checksums are used to limit the memory used to replicate. Later work [46, 47, 20] introduced the idea of using the checksums to detect faults, then using a lineage-like reexecution at various levels, from a function call, to an iteration within a function, to backup.

Checkpointing represents a more complex model for resilience in which copies of application state are made during execution and these copies enable applications to be reexecuted from checkpointed state. Checkpointing often requires intimate application knowledge to accurately capture process state and novel techniques to reduce storage overhead. This is important as, in large systems, with thousands of processors, checkpoint data alone may exceed tens

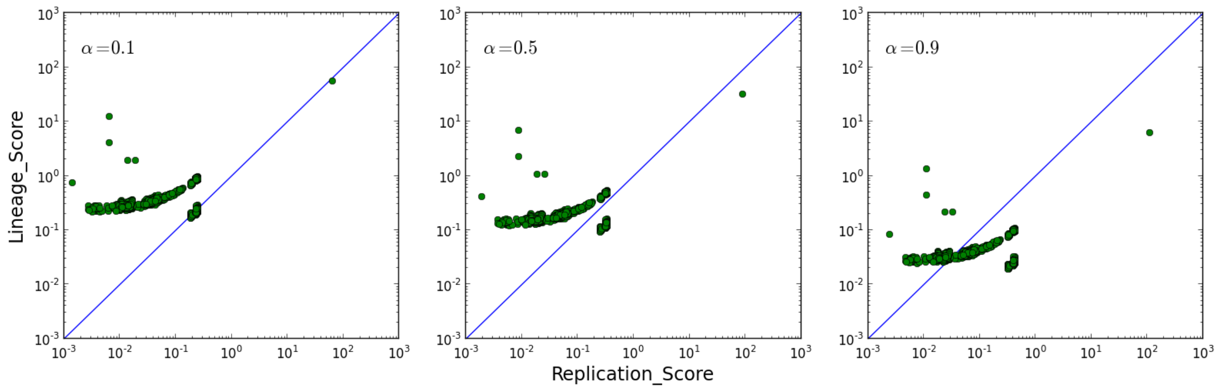

Figure 8: Backup decision making when varying the optimization weight. Files in the upper left triangle are backed up through replication and files in the lower right triangle are backed up through lineage.

of terabytes [18]. For this reason, many different checkpointing approaches have been proposed. In general, approaches can be categorized as either application-, user-, or kernel-level. Application-level approaches are generally one-off implementations, highly optimized based on low-level application knowledge; while highly efficient, development is often expensive. User- and kernel-level checkpointing enable applications to be checkpointed (periodically) during execution without requiring application modifications. In these models, users are often only minimally aware (in some cases they can provide 'hints') of the checkpointing process as the underlying system captures complete application state to reexecute the applications. User level applications include Condor [29] and libckpt [35]. Kernel-level checkpointing is implemented at the operating system, common examples include Distributed MultiThreaded Checkpointing [3] and Berkeley Lab Checkpoint/Restart [21]. Checkpointing-based approaches have also been used to create resilient distributed programming runtimes such as MPICH-V [7] and rMPI [15]. There are other checkpointing research directions as well. Optimum checkpointing interval [48, 11] minimizes the lost work in case of a failure. The consistent checkpointing [14, 10] synchronizes the checkpointed states of each individual process. Checkpointing is typically used on long running tasks, while the tasks in our research are small tasks which can be seen as a natural partitioning of long running task. The long running parallel task abstraction also requires the processes to coordinate upon failure to reach consensus then all processes roll back to the coordinated checkpoint. This is not the case for small task abstraction. The whole execution is usually not halted or rolled back in a collective way, rather the execution continues when failures present, and the recovery decision is making by individual task rather than overall application.

In most cases both replication and checkpointing use stable (often distributed) file system storage of replicas, since these systems are resistant to processor failure. Researchers have investigated the use of disk in RAMCloud [31], diskless [36, 51], and in memory replicas as a means of reducing the I/O overhead of storing replicas on disk. Others have explored the use of SSDs [18] as a compromise between in-memory and disk-based approaches. These models use a number of techniques to avoid the bottlenecks seen with disk-based checkpointing techniques. Other optimizations including encoding techniques, such as erasure encoding [22, 37], have also been applied to reduce data storage requirements.

While these approaches to fault tolerance have been successfully applied in a number of domains they do not provide the same level of dynamism proposed in this work. Perhaps the most similar approaches are those employed in data location aware scheduling [30, 34]. In these systems, the goal is to balance the tradeoff between the cost of transfer and the cost of compute. In such models, sophisticated schedulers determine if data should be moved to compute or vice versa. In many ways these comparisons are similar to the dynamic resiliency algorithms applied in our work; however, the approaches differ in that here we consider the latency within networks and the cost of storage, and then compare this to the cost of computing the files themselves. Moreover, our approach attempts to quantify the cost of reexecution of compute against the cost of storage, where the cost of re-execution is accurately known. In location aware scheduling approaches the cost of transfer and execution are generally estimated with the goal of reducing the overall execution time, irrespective of failure.

7. CONCLUSION AND FUTURE WORK

We have proposed a new approach to fault-tolerant distributed computation suitable for the increasingly important class of parallel applications composed of small tasks linked by data flows when using parallel scripting frameworks with transient in-memory file systems.***Zhao: some more text in the previous sentence The core of our approach is a mathematical model of backup and recovery costs, which we leverage to make runtime decisions about how and when to replicate data and metadata. This model, which permits online comparison of the cost of replicating data via either re-execution or access to replicas, takes as input only input file size, output file size, number of replicas, task time-to-solution, network bandwidth, and failure rate. We implemented the model in a parallel scripting framework and presented performance results for an astronomy image processing application, Montage. These results show that our adaptive scheme can recover from failure more efficiently in the case of failure than other methods, while introducing only 2.9% overhead when there are no failures. We also investigated the sensitivity of our model to failure rate, a notoriously inaccurate parameter. We find that for our target application (and, we expect, for many other similar applications), backup decisions are relatively insensitive to failure rate in practice.

Our configurable model permits users to express their preferences regarding the relative weight to be given to re-execution vs. access to cached data as a recovery strategy, and thus the time spent on data replication vs. metadata storage. Such preferences may be used to express, for example, the user's expected failure rates, their interest in rapid failure-free execution, and/or their interest in limiting storage consumption for backups.

In future work, we plan to integrate this adaptive backup model with the Berkeley Data Analytics Stack, particularly with the graph processing framework GraphX [19] and the memory centric storage system Tachyon [28]. We also plan to investigate a wider range of applications, explore extensions to incorporate other constraints such as limited backup storage, and examine the use of asynchronous replication and hierarchical storage.

8. REFERENCES

[1] T. Amjad, M. Sher, and A. Daud. A survey of dynamic replication strategies for improving data availability in data grids. *Future Generation Computer Systems*, 28(2):337–349, 2012.

[2] D. P. Anderson. Boinc: A system for public-resource computing and storage. In *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, GRID '04, pages 4–10, Washington, DC, USA, 2004. IEEE Computer Society.

[3] J. Ansel, K. Arya, and G. Cooperman. Dmtcp: Transparent checkpointing for cluster computations and the desktop. In *IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, pages 1–12. IEEE, 2009.

[4] Amazon EC2 Service Level Agreement. `http://aws.amazon.com/ec2/sla/`.

[5] J. Bent, D. Thain, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and M. Livny. Explicit control in the batch-aware distributed file system. In *NSDI*, volume 4, pages 365–378, 2004.

[6] D. Borthakur. HDFS architecture. `http://hadoop.apache.org/hdfs/docs/current/hdfs_design.pdf`.

[7] G. Bosilca, A. Bouteiller, F. Cappello, S. Djilali, G. Fedak, C. Germain, T. Herault, P. Lemarinier, O. Lodygensky, F. Magniette, V. Neri, and A. Selikhov. MPICH-V: Toward a scalable fault tolerant MPI for volatile nodes. In *Supercomputing, ACM/IEEE 2002 Conference*, pages 29–29, Nov 2002.

[8] R. N. Calheiros and R. Buyya. Meeting deadlines of scientific workflows in public clouds with tasks replication. *IEEE Transactions on Parallel and Distributed Systems*, 25(7):1787–1796, 2014.

[9] J. Casas, D. Clark, P. Galbiati, R. Konuru, S. Otto, R. Prouty, and J. Walpole. MIST: PVM with transparent migration and checkpointing. In *3rd Annual PVM Users' Group Meeting*, pages 7–9, 1995.

[10] K. M. Chandy and L. Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Transactions on Computer Systems (TOCS)*, 3(1):63–75, 1985.

[11] J. T. Daly. A higher order estimate of the optimum checkpoint interval for restart dumps. *Future Generation Computer Systems*, 22(3):303–312, 2006.

[12] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proc. 8th USENIX Symposium on Operating Systems Design and Implementation*, pages 137–150, 2004.

[13] J. Dongarra et al. The international exascale software project roadmap. *International Journal of High Performance Computing Applications*, page 1094342010391989, 2011.

[14] E. N. Elnozahy, D. B. Johnson, and W. Zwaenepoel. The performance of consistent checkpointing. In *Reliable Distributed Systems, 1992. Proceedings., 11th Symposium on*, pages 39–47. IEEE, 1992.

[15] K. Ferreira, R. Riesen, R. Oldfield, J. Stearley, J. Laros, K. Pedretti, and T. Brightwell. rMPI: increasing fault resiliency in a message-passing environment. Technical Report SAND2011-2488, Sandia National Laboratories, 2011.

[16] I. Foster, J. Voeckler, M. Wilde, and Y. Zhao. Chimera: A virtual data system for representing, querying, and automating data derivation. In *14th International Conference on Scientific and Statistical Database Management*, Edinburgh, Scotland, 2002.

[17] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. *ACM SIGOPS Operating Systems Review*, 37(5):29–43, 2003.

[18] L. A. B. Gomez, N. Maruyama, F. Cappello, and S. Matsuoka. Distributed diskless checkpoint for large scale systems. In *Proceedings of 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, pages 63–72. IEEE Computer Society, 2010.

[19] J. Gonzalez, R. Xin, A. Dave, D. Crankshaw, M. Franklin, and I. Stoica. GraphX: Graph processing in a distributed dataflow framework. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, Broomfield, CO, Oct. 2014. USENIX Association.

[20] J. A. Gunnels, R. A. v. d. Geijn, D. S. Katz, and E. S. Quintana-Ortí. Fault-tolerant high-performance matrix multiplication: Theory and practice. In *Proceedings of the 2001 International Conference on Dependable Systems and Networks (Formerly: FTCS)*, DSN '01, pages 47–56, Washington, DC, USA, 2001. IEEE Computer Society.

[21] P. H. Hargrove and J. C. Duell. Berkeley lab checkpoint/restart (BLCR) for Linux clusters. *Journal of Physics: Conference Series*, 46(1):494, 2006.

[22] C. Huang and L. Xu. Star: An efficient coding scheme for correcting triple storage node failures. *IEEE Transactions on Computers*, 57(7):889–901, 2008.

[23] K.-H. Huang and J. A. Abraham. Algorithm-based fault tolerance for matrix operations. *IEEE Trans. Comput.*, 33(6):518–528, June 1984.

[24] J. C. Jacob, D. S. Katz, G. B. Berriman, J. C. Good, A. C. Laity, E. Deelman, C. Kesselman, G. Singh, M.-H. Su, T. A. Prince, and R. Williams. Montage: a grid portal and software toolkit for science-grade astronomical image mosaicking. *Intl. J. of Comp. Sci. and Eng.*, 4(2):73–87, 2009.

[25] D. S. Katz, T. Armstrong, Z. Zhang, M. Wilde, and J. Wozniak. Many task computing and Blue Waters. Technical Report CI-TR-13-0911, Computation Institute, University of Chicago, November 2011.

[26] D. S. Katz, J. Daly, N. A. Debardeleben, M. Elnozahy, B. Kramer, L. Lathrop, N. Nystrom, K. Milfeld, S. Sanielevici, S. Cott, and L. Votta. 2009 fault tolerance for extreme-scale computing workshop, Albuquerque, NM - March 19-20, 2009. Technical Report ANL/MCS-TM-312, Argonne National Laboratory, December 2009.

[27] D. S. Katz, J. C. Jacob, G. B. Berriman, J. Good, A. C. Laity, E. Deelman, C. Kesselman, and G. Singh. A comparison of two methods for building astronomical image mosaics on a grid. In *Proc. 2005 Intl. Conf. on Par. Proc. Works.*, pages 85–94, 2005.

[28] H. Li, A. Ghodsi, M. Zaharia, E. Baldeschwieler, S. Shenker, and I. Stoica. Tachyon: Memory throughput I/O for cluster computing frameworks. In *7th Workshop on Large-Scale Distributed Systems and Middleware (LADIS'13)*, 2013.

[29] M. J. Litzkow, M. Livny, and M. W. Mutka. Condor-a hunter of idle workstations. In *8th International Conference on Distributed Computing Systems*, pages 104–111. IEEE, 1988.

[30] R. McClatchey, A. Anjum, H. Stockinger, A. Ali, I. Willers, and M. Thomas. Data intensive and network aware (diana) grid scheduling. *Journal of Grid Computing*, 5(1):43–64, 2007.

[31] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum. Fast crash recovery in ramcloud. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 29–41. ACM, 2011.

[32] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, G. Parulkar, M. Rosenblum, et al. The case for ramclouds: scalable high-performance storage entirely in dram. *ACM SIGOPS Operating Systems Review*, 43(4):92–105, 2010.

[33] D. A. Patterson, G. Gibson, and R. H. Katz. A case for redundant arrays of inexpensive disks (RAID). *ACM SIGMOD Record*, 17(3), 1988.

[34] A. D. Peris, J. Hernandez, E. Huedo, and I. M. Llorente. Data location-aware job scheduling in the grid. application to the GridWay metascheduler. *Journal of Physics: Conference Series*, 219(6):062043, 2010.

[35] J. S. Plank, M. Beck, G. Kingsley, and K. Li. Libckpt: Transparent checkpointing under unix. In *Proceedings of the USENIX 1995 Technical Conference Proceedings*, TCON'95, pages 18–18, Berkeley, CA, USA, 1995. USENIX Association.

[36] J. S. Plank, K. Li, and M. A. Puening. Diskless checkpointing. *IEEE Transactions on Parallel and Distributed Systems*, 9(10):972–986, 1998.

[37] J. S. Plank, J. Luo, C. D. Schuman, L. Xu, Z. Wilcox-O'Hearn, et al. A performance evaluation and examination of open-source erasure coding libraries for storage. In *FAST*, volume 9, pages 253–265, 2009.

[38] I. Raicu, I. Foster, and Y. Zhao. Many-task computing for grids and supercomputers. In *Proc. of Many-Task Comp. on Grids and Supercomputers, 2008*, 2008.

[39] I. Raicu, Z. Zhang, M. Wilde, I. Foster, P. Beckman, K. Iskra, and B. Clifford. Toward loosely coupled programming on petascale systems. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, pages 22:1–22:12, Piscataway, NJ, USA, 2008. IEEE Press.

[40] L. Ramakrishnan, C. Koelbel, Y.-S. Kee, R. Wolski, D. Nurmi, D. Gannon, G. Obertelli, A. YarKhan, A. Mandal, T. M. Huang, K. Thyagaraja, and D. Zagorodnov. VGrADS: Enabling e-science workflows on grids and clouds with fault tolerance. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 47:1–47:12, New York, NY, USA, 2009. ACM.

[41] S. M. Rumble, A. Kejriwal, and J. K. Ousterhout. Log-structured memory for dram-based storage. In *FAST*, pages 1–16, 2014.

[42] Snir, Marc. Resilience at Exascale. `http://web.engr.illinois.edu/~snir/PDF/UWM-resilience.pdf`.

[43] G. Stellner. Consistent checkpoints of PVM applications. In *Proceedings of the First European PVM User Group Meeting*. Citeseer, 1994.

[44] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proc. of ACM SIGCOMM '01*, August 2001.

[45] R. Tolosana-Calasanz, J. Á. Bañares, P. Álvarez, J. Ezpeleta, and O. Rana. An uncoordinated asynchronous checkpointing model for hierarchical scientific workflows. *Journal of Computer and System Sciences*, 76(6):403–415, 2010.

[46] M. Turmon, R. Granat, and D. S. Katz. Software-implemented fault detection for high-performance space applications. In *Proceedings of the 2000 International Conference on Dependable Systems and Networks (Formerly FTCS-30 and DCCA-8)*, DSN '00, pages 107–116, Washington, DC, USA, 2000. IEEE Computer Society.

[47] M. Turmon, R. Granat, D. S. Katz, and J. Z. Lou. Tests and tolerances for high-performance software-implemented fault detection. *IEEE Trans. Comput.*, 52(5):579–591, May 2003.

[48] J. W. Young. A first order approximation to the optimum checkpoint interval. *Communications of the ACM*, 17(9):530–531, 1974.

[49] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

[50] Z. Zhang, D. S. Katz, T. G. Armstrong, J. M. Wozniak, and I. Foster. Parallelizing the execution of sequential scripts. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 31:1–31:12, New York, NY, USA, 2013. ACM.

[51] G. Zheng, L. Shi, and L. Kale. FTC-Charm++: an in-memory checkpoint-based fault tolerant runtime for

Charm++ and MPI. In *Proceedings of the IEEE International Conference on Cluster Computing*, pages 93–103, Sept 2004.