Octopus: Efficient Data Intensive Computing on Virtualized Datacenters

Svitlana Tumanova
University of Toronto
svitlana@cs.toronto.edu

Olga Irzak
University of Toronto
oirzak@cs.toronto.edu

Lili Sun
University of Science and Technology Beijing
University of Toronto
lily@cs.toronto.edu

Shiri Margel
University of Toronto
shiri@cs.toronto.edu

Eyal de Lara
University of Toronto
delara@cs.toronto.edu

ABSTRACT

Virtualization provides many benefits for running data intensive workloads including security, performance isolation, and ease of management and configuration. Unfortunately, current VM technology prevents taking advantage of sharing opportunities, resulting in substantial network traffic and application slowdown. Octopus is a new framework for running data intensive applications on virtualized datacenters. Octopus provides efficient file sharing across VMs running on the same physical host and optimizes the placement of VMs in the cluster to maximize sharing opportunities. Our experiments with a suite of bioinformatics and natural language processing applications show that Octopus reduces network transfer by up to 83% and total runtime by up to 55%.

Categories and Subject Descriptors

D.4.3 [**Operating Systems**]: File Systems Management—*Access Methods*; D.4.1 [**Operating Systems**]: Process Management—*Scheduling*

General Terms

Algorithms, Performance

1. INTRODUCTION

The success of cloud computing and improvements in virtualization technology that provide near-native performance have made virtualized datacenters into an attractive solution for running data intensive workloads [20]. Virtualization provides many benefits for high performance data intensive computing including security, performance isolation, ease of management, and the flexibility of running in a custom environment configured with the specific OS version and all necessary libraries. Unfortunately, the strong isolation provided by virtualization also limits the efficiency with which workloads that operate on the same files can be supported.

In traditional native clusters, the underlying operating system leverages its buffer cache to deliver efficient data reuse among multiple jobs. In contrast, when different jobs are each encapsulated in their own VM, there is no opportunity for data reuse as each VM keeps it own independent buffer cache. The result is wasted bandwidth as additional copies of the data need to be transferred over the network, and inefficient use of memory due to duplication when multiple VMs that operate on overlapping files are collocated on the same host.

Opportunities for data reuse exist in a variety of data intensive workloads. For example, a study of the access patterns of jobs running on the cluster that supports Microsoft Bing shows that 18% of the data is accessed by at least three concurrent Scope[1] jobs [7]. Similarly, Ananthanarayanan *et al.* [8] report in their analysis of production workloads from Facebook, Bing and Yahoo! that only 11%, 6%, and 7% of files are accessed a single time, respectively. In addition, a recent trend in the physical and life sciences is to publish large datasets on the cloud giving access to the broad research community to compute on them. For example, Amazon offers free hosting for a growing collection of large public datasets from a variety of fields such as biology, astronomy and geology [6]. Scientists from around the world can explore these datasets by running their applications encapsulated inside VMs.

This paper describes Octopus, a new framework for running data intensive applications on virtualized datacenters. Octopus is designed for I/O bound applications for which data download comprises non-negligible part of application runtime and good performance depends on being able to fit the dataset in memory. The traditional approach for these applications is to subdivide the datasets into smaller entities that fit into the memory of individual nodes, and process them in parallel. For these kinds of applications, increasing the ratio of memory to compute power (i.e., memory per

[1]Scope is a modified form of Dryad[14]

CPU) has a direct effect on performance. Octopus improves application runtime using three techniques: First, it caches files in memory; Second, it enables zero-copy file sharing among VMs running on the same physical host; and Third, it improves locality by collocating on the same physical host multiple VMs (from the same or different users) that operate on common files.

Octopus consists of two mechanisms: Octopus-FS (OFS), a virtualization-aware file system designed for read accesses to large files that supports cross-VM buffer cache sharing; and Octopus-Scheduler (OSched), a data-aware scheduler that improves cluster efficiency by optimizing the placement of VMs to maximize sharing opportunities. OSched has the dual role of job allocator and cluster-wide cache manager, adjusting file distribution to match the dynamic demands of the workload. OSched determines, for every job it schedules, whether to host the job's input files in OFS' shared buffer or the VM's private memory. The former significantly reduces I/O load on the cluster by leveraging locality of reference, and due to files potentially being mapped by multiple VMs, can create a situation where the cumulative memory available to all VMs greatly exceeds the memory of the physical host. The latter allows VMs to transparently access unpopular files without polluting the cache, maximizing reuse and improving overall throughput.

We implemented Octopus as an extension to Xen, and evaluated its performance using a suite of applications from the domains of bioinformatics and natural language processing. We evaluated Octopus using two different methods: First, we validated the correctness of the implementation and measured its performance and overheads by running experiments on our in-house cluster. Second, we used emulation[2] on EC2 to evaluate the performance of Octopus on larger and faster hardware, as well as to explore the sensitivity of the approach to cache size and variations in file popularity. Our experiments show that Octopus reduces cluster-wide network transfer by up to 83% and total runtime by up to 55%.

2. OCTOPUS

Octopus is a framework for data intensive computing on virtualized datacenters. Octopus is optimized for scenarios where datasets are reused by multiple applications or by multiple queries run by the same application. Under these conditions, Octopus reduces network usage, lowers file system load, and decreases application runtimes through on-machine file caching, cross-VM file sharing, and data-aware job scheduling.

Octopus consists of two main components: the Octopus File System (OFS) and the Octopus Scheduler (OSched). OFS creates a buffer cache shared between VMs on a host. It populates the shared cache on demand by fetching the requested files from a traditional network file system, such as NFS or Lustre. OSched builds upon the OFS caching system to provide high levels of data locality for the applications it runs. OSched has functionality for file placement in caches of machines across the cluster, and for scheduling applications. In addition, for every job it schedules, OSched decides whether to run it using OFS or a traditional network file system. By using OFS, OSched reduces I/O load on the cluster and saves memory by eliminating the redundant caching of files on individual VMs. Conversely, by bypassing OFS in favor of a traditional network file system and holding the file on the guest VM's buffer cache, OSched can transparently access unpopular files without polluting the cache, maximizing reuse and improving overall throughput.

[2]OFS requires modifications to the VMM and privileged domain that are currently not supported by EC2.

In order to add a file to the cache, OSched launches an application which uses the desired file on the machine. The application is chosen from the list of applications queued for execution in the cluster. Once the application references a file through OFS, the file is fetched to the machine and stored in cache. In order to remove a file from cache, OSched issues a direct request to OFS.

The OFS caching mechanism works at the file granularity. This approach simplifies the implementation and reduces mapping, communication, and space overhead. On the downside, it puts an upper limit on the maximum file size that OFS supports, i.e., the file has to fit in cache of an individual cluster node. We consider that this restriction is acceptable for our domain, where the established practice is to partition large datasets into smaller files that fit on the memory of individual nodes in order to avoid swapping. Notwithstanding, we intend to explore on demand fetching as part of future work.

We require that the dataset OFS operates on be read only. Neither the scheduled jobs nor external agents may make changes to the data during the runtime of the system. New files, however, may be added to the system at will. This is suitable for a large number of scientific and engineering datasets from fields such as bioinformatics, astronomy, etc. For example, gene sequences and protein structure do not change once discovered[3].

Our scheduling algorithms were designed for independent computational units, which may be different applications, or tasks of an embarrassingly parallel computation. In particular, we allow limited rearrangement of the queue to take better advantage of data locality. Adding support for interdependent communicating applications, e.g., MPI jobs, is intended as future work.

The only requirement on the applications in order to run in Octopus is to notify OSched of the large files it intends to use and to take the paths to those files as command-line parameters. This information allows the scheduler to potentially place the application on the machine with the needed files already in cache and to make a decision as to whether to store the files in cache or not. OSched enforces these decisions by notifying OFS which files the VM is allowed to access.

Presently OSched operates under the assumption that each job has only one input file. The system handles multiple input files by bundling them up into a new virtual file.

We implemented Octopus by extending the Xen hypervisor. OFS requires modifications to the guest and host kernels, as well as small changes to the hypervisor. OSched requires full configuration knowledge of all machines in the cluster and root access to all privileged domains for VM execution and cache management. In addition, guest VMs need to execute a daemon in privileged mode for communication with OSched.

Our system is robust and was built with failure scenarios in mind. OSched has a wrapper around every launched

[3]Sequence error corrections do occur, but are infrequent.

application that reports back whether the application has successfully finished or failed. In particular, the wrapper notifies of a failure in case of file fetching error by OFS. OSched reacts to this case by reexecuting the application using a traditional network file system. In addition, OSched has a monitoring component that periodically polls the machines and individual VMs to ensure they are still alive. The monitoring component also enforces a maximum runtime per application. In case of application failure or timeout, the appropriate VM is discarded and the failure is reported to the user. In case of VM failure, OSched's view of cache contents is adjusted to reflect reality and the respective application gets reexecuted. Finally, in the most severe case of machine failure, the respective machine is taken out of the scheduling pool and the applications that were running on the failed machine get reexecuted elsewhere.

3. OCTOPUS-FS

OFS provides a dynamic caching and sharing layer to a file server, where the files reside permanently. It runs on compute nodes and has a caching and sharing component on the privileged domain (Dom0) and a file system interface on the guest domains (DomUs). The components on the guest and privileged domains communicate via a split driver, while sharing among VMs is built on top of the Xen grant tables mechanism. The OFS architecture is shown in Figure 1.

OFS caches files on the privileged domain by hosting them on RAMFS, which provides a low overhead file system layer on top of the Linux buffer cache. Using RAMFS for storage guarantees the pages stay in RAM and are not swapped to disk. Hosting on the privileged domain means that files can be kept cached in memory across DomU instantiations.

When a file is opened by a guest domain, the request gets relayed to Dom0 which checks if the file is in its cache. If the file is not available, it is fetched from a file server and placed in the cache, potentially evicting an unused file to free the space. Once the file is in cache, Dom0 shares the file's pages with the requesting domain, which then maps the pages into its buffer cache. From that point on, the guest domain may perform read-only file system operations on that file, such as read and mmap without further communication with Dom0. We guarantee that the file's pages will remain in memory for as long as they are used by any guest domain.

A VM can map a file from the shared buffer cache and thus have access to additional memory for file caching purposes. This region may be larger than the memory of the VM itself, and in such cases memory mapping this file is particularly effective. Moreover, the same region can be mapped multiple times in multiple VMs, thus creating a situation where the cumulative memory appears larger than the actual physical memory.

Once a guest domain finishes its work on the file (either the file is closed or the VM is destroyed), the pages get removed from the guest's buffer cache. The usage counter is reduced for the given file, and it may be evicted once the usage reaches zero.

The following sections provide details of the various OFS subcomponents and quantify OFS's performance with a micro benchmark.

3.1 On the Guest Domain

OFS presents DomU with a regular file system interface. Upon mounting OFS, the guest domain gets a full view of

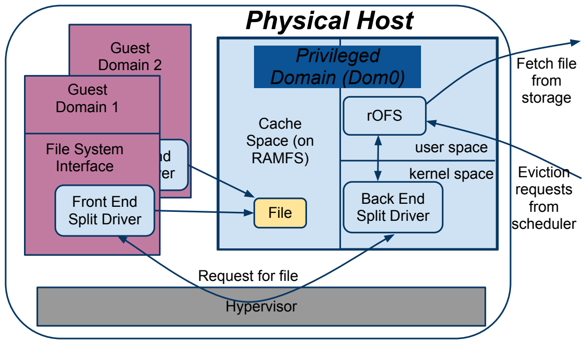

Figure 1: OFS Architecture

the directory and file structure of the remote file system, regardless of whether the files are present on the physical machine or not. Depending on the size of the file system, we may store the representation in the guest domain, Dom0 or populate the directory from the file server on every directory access, thereby trading off space and time overheads. On file open, a request for the file access goes to Dom0. When the file access is granted, grant references are transferred to DomU and mapped in to populate the file's buffer cache. The page grants may be unmapped on file close or on domain exit. We chose domain exit, since our VMs are launched in order to execute one application which mostly uses the large file, and exits once it's done. We work on a file granularity in order to minimize context switching between the domains and do batch grant map-ins. Locality may also be increased and disk seeks decreased on the server when requesting the entire file at once. On the other hand, working on file granularity prevents us from caching files larger than the cache itself, and may leave portions unused.

3.2 In the Hypervisor

We leverage the grant table mechanism to do the core of the page sharing. Since grant tables were not originally meant to support such large volumes of data, we expanded their potential size and added a possibility to grant a page to all VMs that are running on the host. Hence, if the data is public, the grant table needs to have only as many entries as pages in the cache for this use. In Xen 3.4.1, this means (1/512)th of the size of the cache when the cache is fully shared. Memory overhead in Dom0 is one *grant ref* per page. Memory overhead in DomU is a *grant ref*, *struct page* per page and *addr* per mmaped page - still a small fraction of a page. We can reduce the space overhead further by forcing the grant table entries to be consecutive - the overhead for Dom0 and DomU to store *grant refs* drops to two (first and last) per file rather than a *grant ref* per page, though this might result in our grant tables being fragmented.

3.3 On the Privileged Domain

The privileged domain has a kernel level driver, which enables file sharing between guest VMs, and a user level component (rOFS), which fetches the remote file and caches it on the host. The kernel component receives DomU's requests for files, finds the files' pages, grants them and communicates the results back to the guest domain. When Dom0 receives a request for a file which is not in the cache, it communicates it to rOFS which attempts to fetch the file.

The fetch protocol and file server location are configurable parameters provided to rOFS. For performance reasons, protocols are limited to those that can write into RAMFS, but not synchronize to disk. In our setup, we used ftp that writes into RAMFS. Where security is an issue scp is an easy alternative.

rOFS is flexible in handling multiple outstanding requests to different servers. There are tunable parameters which may be varied based on number of servers, network interfaces, disks in the servers, etc.

Cache eviction policy is also controlled by rOFS. It may be configured to use a standard eviction policy such as LRU, LFU, etc, or more interestingly, yield control to the scheduler and execute its directions, since it has better insight into future requests. A file may be removed from the cache and its space freed only when no VM is using it.

3.4 Performance

Figure 2 plots the latency for sharing a file that is present in the cache of Dom0 with an application running on DomU. The measurement includes the time from the point when the application on DomU requests the file until it can access all of its contents. For comparison, we present a RAM to RAM copy of a file within the same domain on the same host. In the absence of page sharing mechanism, a file system will need to copy the pages and RAM to RAM copy is the best case measurement.

The micro benchmark runs on Xen 3.4.1 on a Sun server with 8 cores and 16GB of RAM. Figure 2 shows that OFS is 10 times faster than copying. For large files, RAM to RAM copying achieves a rate between 825 and 845 MB/s, while OFS is in the range of 8533 to 9630 MB/s. A more realistic case than RAM to RAM copy is an NFS server on Dom0 and a client on DomU. When the files are in RAM on Dom0, we measured a 9-fold slowdown compared to RAM to RAM copy, achieving transfer rates of 91 to 94 MB/s for large files. We attribute this performance degradation to the need to copy 3-fold for intra-machine Xen networking and the resulting context switching and additional computation.

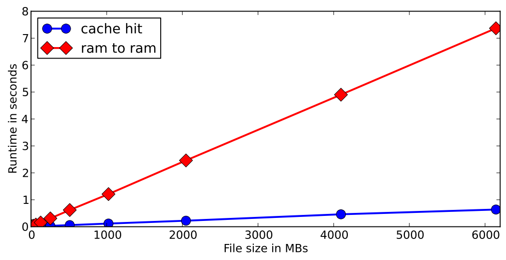

Figure 2: OFS Cache Hit versus RAM to RAM Copy

A cache miss occurs when the file is not present on the host and must be fetched from the file server. We measure the overhead of receiving the request, fetching the file from the server, and sharing the file with the guest domain. The host is connected to the server over a 1Gb network, and the network and server are otherwise idle. Even in the best case, where the file is in the server's memory, the average time to read the remote file was over 50 times slower than a cache hit. This quantifies the potential gains from avoiding duplicate downloads.

Overall, in our experimental setup requesting a file which is already on the host saves us 99% of the time compared to fetching from a warm server for files 128MB or larger. In the case of a miss, we add a modest 1.3% overhead to reading a file over the network.

4. OCTOPUS SCHEDULER

OSched performs three functions: (1), it selects the next job to run and launches it; (2), it manages the allocation of files to caches; and (3), it determines whether a job will access its data over OFS or use a network file system.

4.1 Job Dispatch

OSched makes a job placement decision once it learns that a computational resource has become available on one of the nodes in the cluster. The scheduler selects the job to execute from the queue according to one of the algorithms described in Section 4.4

To launch the job, OSched ssh'es into Dom0 of the respective machine, starts a new guest VM and places the application launch parameters into XenStore, a storage space shared between domains. Once the new guest VM starts, a privileged scheduling daemon reads the parameters from XenStore, and sets up the environment for the job, including mounting OFS if necessary. Next, the daemon launches the application and the wrapper around it which will notify OSched once the job is done or has failed. The average launch time in our system is 18.2 ± 15.8 seconds.

Once OSched receives a notification that a job has finished, it logs the event, ssh'es into Dom0 of the respective node and performs cleanup operations, including destroying the guest VM. Now a computational resource has become available, and the job picking process repeats. The average cleanup time in our system is 15.1 ± 21.9 seconds.

4.2 Cache Management

OSched manages the contents of the OFS cache on every node in the cluster. For every file in cache, the scheduler maintains a state machine and controls state transitions between: "being added", "running", "standby", and "removed."

OSched operates within the constraints of machine cache sizes and sizes of the jobs' input files. It obtains the former from a machine configuration file, which includes additional information such as machine name and the respective number of computational resources. It obtains the latter by polling the file server for sizes of files used by queued jobs.

OSched adds the file to the cache by launching one of the queued applications which uses the respective file as input. Upon a file open request by the application, the file is fetched to the machine's buffer cache as described in Section 3. The fetched file's state transfers from "being added" to "running" as soon as the job is launched. The file stays in "running" state while there is at least one job running on the machine that uses the file. Once the last job using the file on the machine finishes, the file transfers to the "standby" state.

OSched removes the file from cache by explicitly issuing a request to OFS on the respective machine. Once the request returns successfully, the cache entry for the removed file is erased. Cache eviction happens whenever OSched wants to place a new file on a node that meets the following two conditions: (1), the node does not have enough free cache space to hold the file; and (2), the node has standby files, which can be removed to free up space since they are not currently used by any running jobs. We use a FIFO cache eviction policy.

4.3 Data Access Method

OSched determines whether a job will access its files using OFS or a traditional network file system. By instructing the job to use OFS, OSched can reduce I/O load on the cluster by leveraging locality of reference, and can save memory by eliminating the redundant caching of files on individual VMs. On the other hand, by instructing the job to use a traditional network file system (and by extension store the file on its private buffer cache), OSched can run jobs that access unpopular files without polluting the shared cache, as well as schedule jobs on nodes that have idle CPU but not enough free space on the shared buffer cache.

Since OSched requires the applications to take the paths to the large input files as command line parameters, according to algorithm's decision, the scheduler uses the file paths either in OFS or network file systems.

4.4 Scheduling Algorithms

The general problem of optimal file allocation and job scheduling is NP-hard, so we present a number of greedy scheduling algorithm options which approximate the optimal case.

The objective of our scheduling algorithms is to minimize the following two metrics: *network bandwidth*, the total data amount transferred over the network from file servers to cluster machines, and *queue runtime*, the total time it takes for every job in the queue to finish running.

To achieve these goals, OSched collocates computations with cached data, evens out the load across machines through smartly distributing the files in the cluster, and strategically picks the jobs off the queue.

The rest of this section describes how OSched makes file allocation decisions, determines the order of job execution, and sets the frequency with which file allocation decisions are reevaluated.

4.4.1 File Allocation

The file allocation strategy assigns a portion of the cluster-wide cache to individual files according to their *fair share*, defined in terms of the file's popularity among jobs in the queue and the size of the file.

The goal of the algorithm is to bring the *current share*, defined as the current file distribution in the cluster, closer to the fair share allocation, while accounting for the future demand originating from queued jobs and the current state of the cluster.

OSched supports 3 options for defining file popularity and calculating the fair and current shares: *Replica*, *Running*, and *Standby*. Replica takes into account only the number of times the file was replicated across machine caches. Running also takes into account how many jobs within the machines are using the file. Finally, Standby also accounts for the standby file instances. Figure 3 shows the formulas for calculating fair share (FS) and current share (CS) for each approach. For the abbreviations used in formulas, please, see Table 1.

Each approach has its strengths and its weaknesses. Replica is the most conservative and stable, but it doesn't take into account how much file is shared within the machine.

Table 1: File Allocation Calculation Abbreviations

FS	fair file share allocation
CS	current file share allocation
M	# machines
F	# files
J	# jobs
q(*file*)	# jobs in queue using *file*
r(*file*)	# running jobs in cluster using *file*
st(*file*)	# standby replicas of *file* in cluster
total data	total size of all files used by queued and running jobs
total standby	total size of all files in standby caches

$$CS_{replica}(file) = \#\ file\ replicas\ across\ cluster$$

$$FS_{replica}(file) = \frac{(q(file) + r(file)) \times size(file)}{total\ data} \times M$$

$$CS_{running}(file) = \frac{r(file) \times size(file)}{total\ data}$$

$$FS_{running}(file) = \frac{(q(file) + r(file)) \times size(file)}{total\ data}$$

$$CS_{standby}(file) = \frac{(r(file) + st(file)) \times size(file)}{total\ data + total\ standby}$$

$$FS_{standby}(file) = \frac{(q(file) + r(file) + st(file)) \times size(file)}{total\ data + total\ standby}$$

Figure 3: Fair and Current Share Formulas

Standby makes use of the most information about the cluster, but is fairly volatile and can potentially lead to thrashing. Running is the middle ground between the above two options.

The algorithm goes as follows:

```
(1) calculate fair share allocation
(2) for every file that fits into cache
(3)    calculate current share as if the job
         for the file was put on machine
(4)    calculate diff with fair share
(5) pick files with min diff
(6) break the ties
(7) pick first job from queue for chosen file
```

We illustrate the file allocation algorithm using a simple example with two machines (See Figure 4). Machine 1 has two jobs running on it, both of which are using file A. In addition, Machine 1 has file C in standby. Machine 2 has one job running on it, using file B. The queue contains two jobs using files A and C respectively. The sizes of the files are 1. The goal of the example is to determine which file to load into the cache of Machine 2, and, by extension, which of the two queued jobs to run on the available processor on Machine 2.

Step (1). Using Replica, let's compute the fair share allocation.

$$FS(A) = \frac{(1+2) \times 1}{5} \times 2 = \frac{6}{5}$$

Similarly,

$$FS(B) = \frac{2}{5}, FS(C) = \frac{2}{5}$$

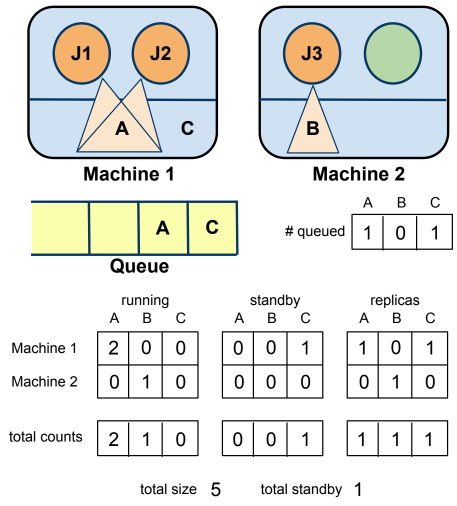

Figure 4: Share Allocation Example

. Then fair share allocation is $(\frac{6}{5}, \frac{2}{5}, \frac{2}{5})$.

Steps (2) - (3). Assuming that both of the files A and C can fit into cache, we need to calculate the current allocation for both of them. The current allocation, if the job for file A was put on Machine 2, is:

$$CS(A) = 2, CS(B) = 1, CS(C) = 1 \Rightarrow (2, 1, 1)$$

Step (4): The difference between the current and fair share allocations is defined as the sum of the squared differences of the shares of every file. Thus:

$$diff_A = (\frac{6}{5} - 2)^2 + (\frac{2}{5} - 1)^2 + (\frac{2}{5} - 1)^2 = 1.36$$

Similarly, if job for file C was put on Machine 2, the current allocation would be $(1, 1, 2)$ with $diff_C = 2.96$.

Step (5). The minimum difference is $diff_A$.

Step (6). In the event of a tie (not the case in this example), we use one of the job picking strategies described in Section 4.4.3.

Step (7). First job on the queue using file A is picked to be scheduled on Machine 2. Once the job accesses input file A, the file will be brought into cache of Machine 2.

4.4.2 *Recalculation Frequency*

OSched supports two modalities that control the frequency with which file allocations get recalculated. *Eager*, which forces a recalculation every time a new job gets scheduled, and *Lazy*, which triggers recalculation only when it's necessary to bring a new file on machine. Lazy runs as follows: most of the time it uses one of the job selection options described in Section 4.4.3 to pick a job for a file already present on the machine. Once there are no more jobs in queue for files on machine, Lazy forces a recalculation.

Eager has an advantage of being as close as possible to the fair share file distribution. However, due to adjusting to the fair share allocation, it might be foregoing data locality opportunities. Lazy, on the other hand, provides the best data locality results; however, it can diverge from the fair share allocation substantially over periods of non-recalculation.

4.4.3 *Job Selection*

Job selection strategies are invoked to decide which job to pick among the ones using the given set of files. The situation arises either when we need to break the ties between eligible files in Section 4.4.1 or when we are picking a job for files already present on machine (Lazy algorithm, Section 4.4.2). OSched supports the following job-picking strategies: *Size* - pick the oldest job for the largest file; *Queue* - pick the oldest job for the file that has the biggest number of jobs queued up for it; and *Runtime* - pick the oldest job for a file with longest runtime approximated by $q(file) \times size(file)$.

The described algorithms allow for unbounded rearrangement of the queue, potentially resulting in job starvation. OSched avoids starvation by introducing a K-hop clause: if K jobs, located after this one in queue, were scheduled, the job is force scheduled once the next resource becomes available.

4.4.4 *Bypassing OFS*

There can be a time when the above algorithms do not return the job to schedule, even though there are still jobs in the queue. It can happen, for example, when there is not enough space in the cache to bring in another file in either regular algorithm flow or when the job is force scheduled. To prevent resource idleness, we still run the job, but fetch its files through a traditional network file system, avoiding OFS.

4.4.5 *Putting It All Together: Fair Share Algorithm*

Putting all of the above-discussed algorithm options together, we obtain the following algorithm:

```
once comp resource becomes available:
if (first job in queue was skipped K times)
  force-schedule the job via
    OFS if there is enough space in cache
    NFS otherwise
else
  execute Eager or Lazy algorithm
  if job was returned
    launch job through OFS
  else
    launch first job in queue through NFS
```

4.4.6 *Algorithmic Complexity*

The calculations performed by the file allocation algorithm are supported by two matrices of size $(M \times F)$ which record for each machine the number of currently running jobs using a given file, and the number of files in standby cache. In addition, we use three F-sized arrays containing the number of queued jobs using each file, number of running jobs using each file, and number of file replicas across the cluster. *Total data* and *total standby*, incorporating file sizes into summary values, are maintained as well. Hence, the space overhead of the algorithm is linear in $\#machines$ and $\#files$.

Building those data structures requires an initial cost of $O(F \times (M + J))$. Then, the algorithm iteration cost is $O(F_{st} \times F)$ where F_{st} is the number of standby files on machine with $F_{st} \ll F$. Since only a small subset of files is going to be present on the machine and the number of files should be much smaller than the number of jobs operating on them, we expect the algorithm overhead to be small.

The worst-case runtime complexity of the Eager algorithm is $O(J + F_{st} \times F)$. $O(J)$ part comes from scanning the queue to find a new job to schedule for the files present on machine. In most cases a job will be found before the end of the queue is reached.

5. EVALUATION

We evaluated Octopus using two different methods: First, we validated the correctness of the implementation and measured its performance and overheads by running experiments on our in-house cluster. Second, we used emulation on EC2 to evaluate the performance of Octopus on larger and faster hardware, as well as to explore the sensitivity of the approach to cache size and variations in file popularity.

We answer the following questions:

1. How does Octopus perform, as measured by queue runtime and network traffic, compared to an approach where individual VMs download their data from a network file system and store it on their private buffer cache?
2. When a job cannot run on a node using OFS because of lack of cache space, is it beneficial to run the job using a traditional network file system?
3. How sensitive is Octopus' performance to the ratio of compute and storage nodes in the cluster?
4. How well does Octopus perform under different file distributions?
5. How sensitive is Octopus' performance to the size of the cache?

5.1 In-House Cluster

Our in-house cluster consists of 12 Sun Fire X2250s (each with eight Xeon cores, 8 GB RAM, and one Gigabit Ethernet NIC). One of the machines was occupied by OSched, launching jobs. A second machine was designated as a data server, and was configured with 2 hard-drives, stripped with RAID 0. The remaining 10 machines were used for job execution: every machine could run up to 7 single-CPU guest VMs, and 3.5 GB of virtual memory on each machine was allocated to OFS. The data server ran an FTP server and an NFS server. The FTP server is used by OFS to transfer files into local caches. NFS was used for comparison purposes. We chose to use NFS, as opposed to a more sophisticated HPC file system, such as Lustre, due to the relatively small size of our setup.

5.1.1 Workload

We experimented with the following popular bioinformatics applications: BLAT, FASTA, FSA-Blast, Ssearch, and Tandem Repeats. BLAT, FASTA, FSA-Blast, and Ssearch are all variations of the NCBI-BLAST application which performs a search of protein or nucleotide sequence against a genome. The applications vary in the level of search sensitivity from the most thorough (Ssearch) to least thorough, and, thus, least accurate (BLAT). Tandem Repeats searches for two or more contiguous approximate copies of a pattern of nucleotides within a genome. The applications also vary in the file access pattern: FSA-BLAST memory maps the entire file while the rest of applications read the file sequentially through standard interface.

We used public genomes downloaded from the website of the University of California at Santa Cruz [3] as inputs in our experiments. We had a total of 34 files, ranging in size from 150 MB to 3.5 GB (mean: 1.5 GB, median: 1.4 GB) and adding up to 51 GB in total.

Each job operated on by the Scheduler consists of one of the above applications running a single search query against a single genome. The jobs also include the operations of VM startup, configuration, cleanup, and reporting back to the Scheduler once the application finishes the execution.

The job queues we use in our experiments give each application approximately the same total runtime to mimic the equal part of the cluster given to every user. As such, the fastest-running BLAT was given a 60% allocation, whereas FASTA, FSA, Ssearch and Tandem Repeats were given 23%, 10%, 4% and 3%, respectively.

The job queue consisted of 910 jobs taken from 2 skewed distributions (A and B), and arranged in the queue as ABA. A and B each contained 303 jobs, and had a non-intersecting set of 3 *hot* files used by 50% of the jobs. The scheduler operated with a window size of 303 jobs and used K-hop with the value for K set to 105.

5.1.2 Octopus Performance

The two plots on the left side of Figure 5 show the runtime and network usage under three scenarios running on our in-house cluster: Octopus using exclusively OFS (pure-OFS), Octopus using a combination of OFS and NFS (OFS+NFS), and an approach where individual VMs download their data from the network file system (NFS).

Both pure-OFS and OFS+NFS used the best performing OSched algorithm "eager size replica." NFS uses a scheduling algorithm that optimizes the effectiveness of the NFS server cache by maximizing file locality across the cluster as follows:

```
{keep track of last_file}
if there are jobs in queue using last_file
   pick a first job in queue using this file
otherwise
   pick a fifo job off the queue
   update the last_file
```

The results of the experiment show that pure-OFS sends 83% less data than NFS, which results in a 47.5% reduction in runtime. These results show the advantage of sharing files across guest VMs, and validate Octopus' approach of collocating jobs with their data, and dynamically assigning cluster cache resources to files proportionally to their popularity.

OFS+NFS further reduces runtime by additional 6%, but this comes at a cost of 24% increase in network downloads compared to pure-OFS. This result shows that when it is not possible to run jobs using OFS because of lack of cache space, is it beneficial to leverage otherwise idle CPUs by running jobs using a traditional network file system.

5.1.3 Compute to Storage Ratio

Figure 6 shows the effects on runtime and network usage of varying the ratio of compute to storage nodes in the cluster. The plot shows results for the OFS and NFS algorithms introduced in the previous section. All runs use a single data server, while the number of compute nodes increases from 2 to 10, and the size of the job queue is scaled linearly with the number of compute nodes (up to a maximum of 490 jobs).

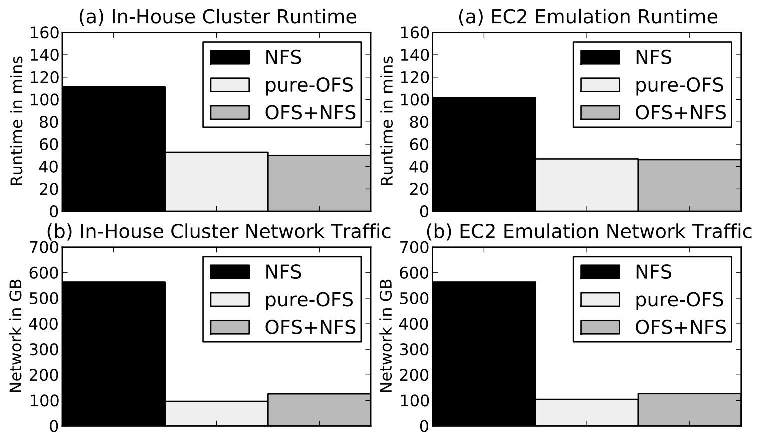

Figure 5: Octopus Performance. OFS greatly reduces network traffic and queue runtime. Performance of EC2 emulation is comparable to in-house prototype.

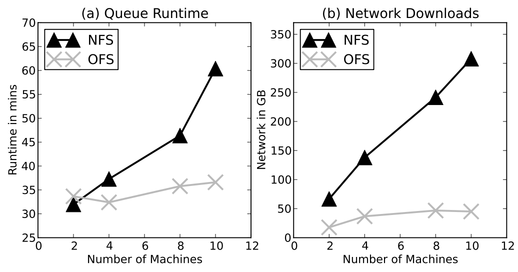

Figure 6: Compute to Storage Ratio Sensitivity. Octopus handles an increase in the number of compute nodes gracefully, whereas NFS' performance degrades rapidly.

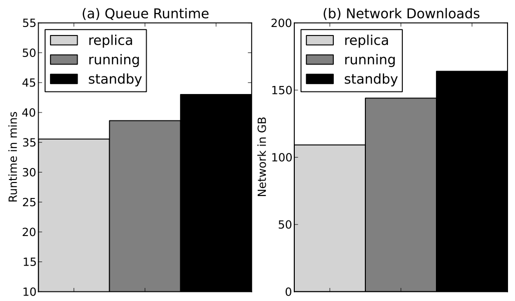

Figure 7: Share Calculation Options Comparison. Replica is most conservative in file redistribution.

Since we are scaling the size of the queue together with the number of machines, we expect the runtime to stay roughly the same if the data server handles the load well. This is indeed the case for Octopus, which successfully handles the increase in compute to storage ratio by leveraging data locality to reduce network load. In contrast, while NFS is able to match Octopus' performance when the compute to storage ratio is low, its performance degrades rapidly as we add compute elements and the file server struggles to handle the increased demand for files. The implications is that in order to match Octopus' performance using a traditional file system, a cloud provider would need to keep the computation to storage ratio at very low level by dedicating (at considerable expense) a large fraction of the cluster to storage.

5.1.4 Scheduler Variations

Figure 7, compares different "fair share" calculation options. The experiment uses "eager" flow and "size" option for job picking. The size of the queue was 700 jobs, with K-hop set to 105. "Replica" performs best in terms both of network traffic and runtime. The approach benefits from its more conservative strategy for file redistribution: adding another replica of the file makes a big difference in the current share calculation, thus, encouraging usage of files already local to a machine.

The different strategies for "job picking" and approaches for setting the "recalculation frequency" had little effect on runtime or network load (around 5%). In summary, the best performing algorithm is "eager size replica," which is the algorithm that we used in all other sections of the evaluation.

5.2 EC2

We conducted experiments on Amazon EC2 to evaluate the performance of Octopus on hardware with more memory. Unfortunately, because OFS requires changes to the hypervisor and privileged domain it cannot currently run on EC2. Similarly, OSched requires control over the placement of VMs to physical hosts, which is also not presently possible on EC2 [4].

Instead, we emulated the required functionality by using different operating system processes within individual EC2 VM instances. In the emulation, every EC2 VM instance corresponds to a physical machine in our in-house experiments, and each vCPU represents an in-house VM.

[4]EC2 cluster placement groups enable customers to place groups of VM on the same rack, but do not provide control over placement on individual physical hosts.

We emulated the functionality of the OFS shared buffer using RAMFS. When an emulated VM opens a file that is not available on the simulated shared buffer cache, the emulated VM blocks while the file is copied from the remote file server into RAMFS. On a cache hit, the emulated VM blocks for the estimated time that it would take to map the memory pages from the OFS cache into the requesting VM. We calculated this time from the measurements presented in Section 3.4.

For consistency, our NFS-based experiments use a similar approach, with multiple emulated VMs running on different vCPUs of the same EC2 instance. To ensure that different jobs executed on the same EC2 instance do not benefit from sharing, which would not be available had each job executed on a separate VM, we changed the file naming scheme to guarantee that each job works on what it perceived to be a unique file.

To validate the emulation we reran the experiment described in Section 5.1.2 using 10 EC2 instances, each running 7 emulated VMs with an emulated Octopus cache size of 3.5 GB. In addition, two more EC2 instances were configured to run OSched and an NFS and FTP file servers. All EC2 instances are of type High-Memory Quadruple Extra Large (m2.4xlarge), with 8 vCPUs, 68.4 GB of memory and High I/O [5].

The plots on the right side of Figure 5 show the results of this experiment. The results on the emulated EC2 setup, both in terms of bandwidth and runtime, are remarkably close to the full-featured system running on our in-house cluster.

5.2.1 Workload

We ran experiments with applications from two domains: bioinformatics and natural language processing. The first one is the bioinformatics FASTA application, which we used in our previous experiments. In our experiments we used 20 different 8.5 GB files, each representing a different genome. The second application, Artist Recognition, is a natural language processing application that identifies the likely performer of a song based on its lyrics. The application consists of separate training and testing tasks. We ran the Artist Recognition application on the Million Song Dataset [2], a collection of audio features and metadata for a million contemporary popular music tracks available as an Amazon Public Dataset. The entire dataset in the experiment is 280 GB. It contains 44,745 unique artists based on their Echo Nest ID, which is split into 26 main downloads (letters A-Z), 11 GB per each.

We worked with a job queue of 900 jobs. Each application got approximately the same total runtime. As such, FASTA (which is heavier than MillionSongs application) got 11% of the jobs (100 jobs) and the MillionSongs training and testing application got 44% of the jobs each (400 jobs). All together we had a dataset of size 456 GB (8.5 GB*20 + 11 GB*26). Moreover, since we worked with cache of size 15 GB, only one file could fit the cache at any point. We worked with a K size of 105 and a window size of 300.

[5]While Amazon does not publish explicit network performance for this kind of instance, we measured typical throughput of 1 Gbps.

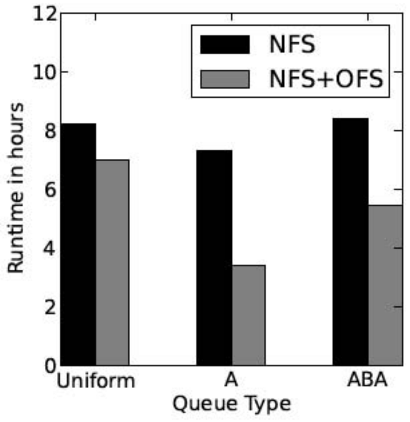

Figure 8: Effect of data locality on performance.

5.2.2 File Popularity

We used three different job queue structures to explore the effects of varying the file popularity.

1. Uniform Probability: Each file appears in the job queue the same number of times. For FASTA each file appears 5 times in average. For MillionSongs each file appears 30-31 times in average.
2. One set of "hot" files (A Queue): Some of the files will appear much more than others - "hot" files. We define D - the percent of the hot files from all of the files and P - the percent of the hot files from the queue. We chose to work with D = 20% and P = 80%. In this case we will have 4 hot files from the FASTA files that will appear 20 times each, and 5 hot files from the MillionSongs files, that will appear 128 times each.
3. Two alternating sets of "hot" files (ABA Queue): Work with the A Queue for the first 300 jobs, then choose new hot files for the next 300 jobs (B), then returns to the A structure for the last 300 jobs.

Figure 8 shows the results of this experiment. As could be expected, the degree to which Octopus improves execution time is a function of the level of data locality in the job queue. The *Uniform* queue presents the least opportunities for reuse, and thus Octopus achieves only a modest 15% improvement in execution time. On the other side of the spectrum, the *A* queue has the greatest potential for reuse leading to an impressive 54% speedup compared to NFS. The potential for data reuse in the *ABA* queue falls in between these two scenarios resulting in a 34% reduction in queue completion time.

5.2.3 Cache Size

In order to explore the influence of the cache size on the efficiency of the computation time and network transfer we repeated the previous experiment with the ABA job queue 3 times - with different cache sizes. In order to change the cache size we changed the RAMFS size. We used RAMFS of size 15 GB, 25 GB and 35 GB. The first cache can fit one file (since the file size is 8.5 GB or 11 GB). The second cache can fit 2 files and the third cache can fit 3 files. Figure 9 shows the relation between the cache size and the overall runtime, network transfer, as well as cache hit rate. As expected, as

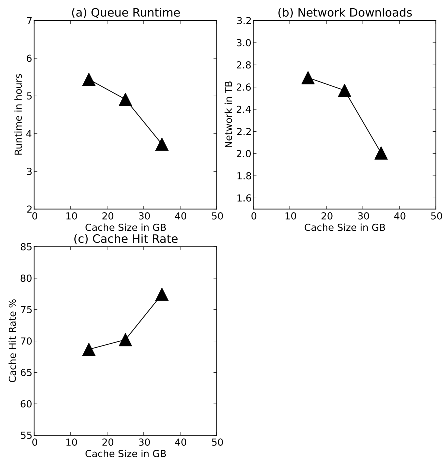

Figure 9: Comparison between the runtime and network transfer of different cache sizes.

we increase the cache size, we increase the cache hit rate and thus both the runtime and the network transfer are greatly improved. We can see that when using cache of size 35 GB, 26% less data is sent in the network which results in a 32% reduction in runtime, compared to cache of size 15 GB.

6. DISCUSSION

In this paper, we explore the premise that a scheduler is more suitable to determine cache usage and eviction than the application, the VM or even the physical machine that hosts the cache. We argue that the scheduler has a more holistic view of the system. The scheduler is aware of the upcoming jobs in the queue, network contention, load and capacity per machine, and data distribution. While we explore taking advantage of this knowledge for distributed cache management, the idea that the scheduler may have the ability to influence application behavior is more general.

For example, an application may have the choice of fetching data from the local disk, or the memory of another machine in the cluster. Assuming a Gigabit network connection, nearby memory is a much faster approach than local disk. However, it is not the case when the nearby machine is overloaded, or when network contention is high. With dynamic hotness, this situation may not always be mitigated by better data placement. Hence, fetching decisions are better made with realtime information about the system state - the information which the scheduler has.

Memory allocated to the Octopus cache comes at the cost of reducing the pool of memory available to individual VMs, and can even limit the number of VMs that can be instantiated. Thus, the optimal size of the Octopus cache depends on several system factors, particularly, the scarceness of RAM versus sharing opportunities and file server load.

As part of future work, we will explore dynamically adjusting the footprint of our caching system to reflect memory contention versus sharing opportunities. To accomplish this efficiently, we will add the ability to evict data from the cache to the VM's private buffer cache even while the page is in use and vice-versa. The scheduler's complexity will increase due to the new requirement of detecting the factors affecting the system's efficiency, and of increasing or reducing the size of cache on selected hosts accordingly.

7. RELATED WORK

While there is significant previous research on VM page sharing, as well as data-aware scheduling, to the best of our knowledge, Octopus is the first system that improves the efficiency of data-intensive cloud applications by providing a scheduler-controlled cache that supports efficient sharing of remote data across virtual machines.

Memory de-duplication is supported by the VMware ESX server [21] and has been explored in other research systems, such as the Difference Engine [13]. These systems periodically determine sharing opportunities and consolidate identified pages across VMs. The advantage of this approach is that it identifies all sharing opportunities, rather than only the ones in a given file system. For our use case of data intensive applications, the data needs to be downloaded into the VM, fit in memory for each VM and then the sharing opportunities will be identified. However, typically, the datasets are larger than an individual VM's memory, so swapping will occur. Few sharing opportunities will be identified as a result (since they are identified in memory and not on disk), and those will be transient due to frequent swapping. Moreover, the bandwidth of downloading the datasets for every VM will be many-folds more than necessary. Satori [17] does identify sharing opportunities at the block device layer whenever they are available locally. However, when the files are remote, Satori is provided with few sharing opportunities, since it attempts to condense VM's existing memory, rather than create a shared pool of memory that all VMs can take advantage of.

XenFS [4] promises to provide guest VMs with a unified local file system and shared buffer cache. However, it does not provide users with seamless access to remote files. The guest must explicitly bring the file to the machine and write it into the cache if it is not there. Moreover, guest VMs will need to coordinate what data to bring into the unified cache and what to evict.

We can categorize the existing body of work in data distribution and application scheduling according to whether the data is stored in dedicated file servers or distributed over the compute nodes in the cluster. Systems with a separation between storage and compute nodes can react to changing popularity of the dataset by caching on the compute nodes. Zazen [19] caches files it knows will be input to future jobs and schedules the jobs according to data locality. However, it focuses on a parallel computation pipeline where output from one step is input to another. Our solution is more general since it does not restrict jobs to any programming paradigm and it allows sharing the files between any jobs in the system. The Co-scheduling system [18] adds data-awareness to Condor [1]. The system dynamically caches popular datasets on the worker nodes and schedules jobs according to availability of both CPU and data on a given node. However, this system does not target a virtualized environment, has different caching constraints and does not have two different modes of data fetching which may be used concurrently. BadFS [9] caches remote data on local memory and argues that the scheduler should have control over data transfer and caching. However, it targets a WAN environment, so caching and data transfer are optimized between clusters, as opposed to within clusters.

Distributed storage systems, such as MapReduce [10]/ GFS [11], Hadoop [5], Sector/Sphere [12], and Stork [15] are designed to exploit data locality [22]. However, they replicate their data either by predicting demand or statically for reliability purposes. In contrast, our system's replication strategy involves observing the changes in hotness of files and using the files' current popularity to calculate how much of the cluster to allocate to those files. If popularity is dynamic or hard to predict, as may be the case in multi-user cloud environments, the dynamic caching mechanism provided by Octopus could be used to augment the capabilities of a distributed storage system.

Scarlett [7] is a system that replicates files based on their popularity in MapReduce environment. Similarly, MixApart [16] is a system that allows MapReduce computations to analyze data stored on enterprise storage systems by replicating and caching data on local disk. Octopus differs from both systems in that it caches files in memory and is specially designed to enable sharing in virtualized clusters.

8. CONCLUSION

We set out to improve the performance of data intensive applications that run on virtualized datacenters. Our solution involves caching and sharing datasets via OFS as well as maximizing data locality opportunities via OSched. OFS allows VMs running on the same host to efficiently share files in the buffer cache thereby avoiding duplicate pages and unnecessary downloads. This approach can create a situation where the cumulative memory available to all VMs greatly exceeds the memory of the physical host. OSched builds on top of OFS and performs data-local application placement and popularity proportional file distribution. Our experiments show that Octopus reduces network traffic by up to 83%, and queue completion time by up to 55%. Octopus is ideally suited for environments where the popularity of datasets changes dynamically, and can be used as a caching solution to augment existing network file systems. In the future we plan to integrate OFS with Hadoop to improve the performance of map-reduce jobs on clusters that use virtualization. We also plan to extend Octopus with support for gang scheduling to support the execution of tightly-coupled parallel tasks, and to include support for applications that use multiple input files.

9. REFERENCES

[1] Condor. http://research.cs.wisc.edu/condor/.

[2] Million Song Dataset. http://labrosa.ee.columbia.edu/millionsong.

[3] UCSC Genome Bioinformatics. http://genome.ucsc.edu.

[4] XenFS. http://wiki.xensource.com/xenwiki/XenFS, 2007.

[5] Apache Hadoop. http://hadoop.apache.org, 2011.

[6] Public Data Sets on Amazon Web Services. http://aws.amazon.com/publicdatasets/, 2011.

[7] G. Ananthanarayanan, S. Agarwal, S. Kandula, A. Greenberg, I. Stoica, D. Harlan, and E. Harris. Scarlett: Coping with skewed content popularity in mapreduce cluster. In *EuroSys: Proceedings of ACM SIGOPS/EuroSys European Conference on Computer Systems*, Salzburg, Austria, April 2011.

[8] G. Ananthanarayanan, A. Ghodsi, A. Wang, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica. Pacman: Coordinated memory caching for parallel jobs. In *NSDI: Symposium on Networked Systems Design and Implementation*, San Jose, CA, April 2012.

[9] J. Bent, D. Thain, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and M. Livny. Explicit control in a batch-aware distributed file system, 2004.

[10] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51:107–113, January 2008.

[11] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. *SIGOPS Oper. Syst. Rev.*, 37:29–43, October 2003.

[12] Y. Gu and R. Grossman. Toward efficient and simplified distributed data intensive computing. *IEEE Trans. Parallel Distrib. Syst.*, 22:974–984, 2011.

[13] D. Gupta, S. Lee, M. Vrable, S. Savage, A. C. Snoeren, G. Varghese, G. M. Voelker, and A. Vahdat. Difference Engine: Harnessing memory redundancy in virtual machines. *Commun. ACM*, 53:85–93, October 2010.

[14] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 59–72, New York, NY, USA, 2007. ACM.

[15] T. Kosar and M. Livny. Stork: Making data placement a first class citizen in the grid. ICDCS '04, pages 342–349, Washington, DC, USA, 2004.

[16] M. Mihailescu, G. Soundararajan, and C. Amza. MixApart: Decoupled analytics for shared storage systems. In *Conference on File and Storage Technologies*, San Jose, CA, February 2013.

[17] G. Milos, D. G. Murray, S. Hand, and M. Fetterman. Satori: Enlightened page sharing. In *Usenix*, 2009.

[18] A. Romosan, D. Rotem, A. Shoshani, and D. Wright. Co-scheduling of computation and data on computer clusters. SSDBM, 2005.

[19] T. Tu, C. A. Rendleman, P. J. Miller, F. Sacerdoti, R. O. Dror, and D. E. Shaw. Accelerating parallel analysis of scientific simulation data via Zazen. In *8th USENIX Conference on File and Storage Technologies*, Usenix FAST, 2010.

[20] R. Tudoran, A. Costan, G. Antoniu, and L. Bounge. A performance evaluation of Azure and Nimbus clouds for scientific applications. In *2nd International Workshop on Cloud Computing Platforms*, Bern, Switzerland, April 2012.

[21] C. A. Waldspurger. Memory resource management in VMware ESX server. *SIGOPS Oper. Syst. Rev.*, 36:181–194, December 2002.

[22] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. EuroSys '10, pages 265–278, New York, NY, USA, 2010.