1 VM I/O Pass-through

Most existing cloud workloads run on virtual machines. In order for a file within a VM to be visible to analytics workloads running on other VMs or physical servers, the file's block addresses should be captured and passed through the virtualization layer. Based on the principles discussed in section **??**, `LiveMR` captures changes to the block addresses of an `inode`, and sends such updates to the virtualization layer. As illustrated in Figure 1, this allows the hypervisor to be aware of the offset of a VM local file (e.g., log.txt) in the disk image file. Combined with VFS mapping discussed in section 1.1, this allows the construction of hypervisor local files (e.g., BigLog.txt) mapping to one or multiple VM files. The rest of this section describes the changes we made to `ext4` and `QEMU`.

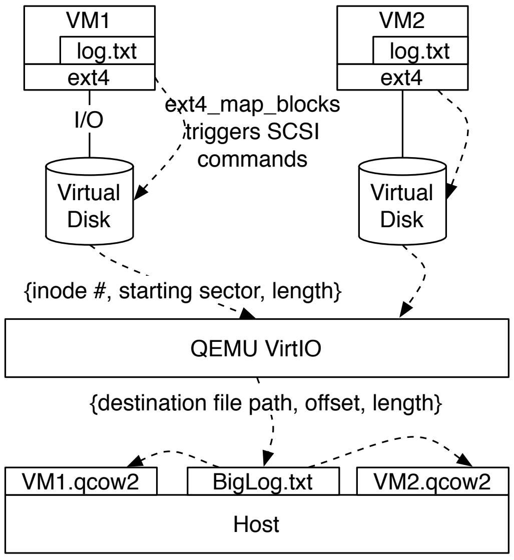
VM1
log.txt
ext4
VM2
log.txt
ext4
I/O
ext4_map_blocks triggers SCSI commands
Virtual Disk
Virtual Disk
{inode #, starting sector, length}
QEMU VirtIO
{destination file path, offset, length}
VM1.qcow2
BigLog.txt
VM2.qcow2
Host

Figure 1: I/O flow with our mechanism

ext4: Linux `ext` file systems store block addresses as an `i_block[]` array in the `inode` (up to 3 levels of indirect addressing is used). Contents of `i_block[]` is changed when the file is expanded, truncated, or in rare cases moved. `LiveMR` captures these 3 events and sends `ioctl` messages to the underlying virtual disk. The most common case are file expansions, where the function `ext4_map_blocks()` allocates free blocks to a file. Whenever `ext4_map_blocks()` returns, `LiveMR` captures the allocated block addresses and sends an `ioctl` command to the block device layer, which in turn composes a `SCSI` command including the `inode` number (4 bytes), the starting sector number (4 bytes), as well as the size of the allocation (2 bytes).

Two `ext4` optimizations are relevant to the design of `LiveMR`. First, `ext4` uses delayed writing by default, leading to delayed block allocations. Dirty blocks from applications are first kept in the page cache, and then flushed to disk based on their age and read requests. In general, this delay is tolerable for analytics applications. Second, `ext4` is an extent-based file system, meaning that each file is allocated a contiguous region on the disk preparing for future growth. This enables `LiveMR` to merge contiguous update requests.

However, as will be shown in section **??**, when direct I/O is used in the VM, the overhead can be as high as 36%. To mitigate this issue, `LiveMR` supports batched metadata updating. A queue for update requests is maintained in the block device layer, immediately before SCSI commands are sent to virtual disk devices and causing context switches for the calling application. Similar to the OS page cache flushing policy, `LiveMR` issues SCSI commands only if: 1) An update has been queued for more than T seconds; 2) The total amount of queued updates is over SMB of size. Because the target workload consists mainly of sequential data appending, we have added another flushing condition: 3) The newly arrived update request is not contiguous with the last update from the same file (and therefore cannot be merged). This simplifies the design, allowing `LiveMR` to queue at most one request for each file.

QEMU: `LiveMR` extends `QEMU` in support of the `SCSI` command sent from the VM. It then translates the sector numbers into offsets in the disk image files, which in the most common case are of `qcow2` format. It then passes the information to the physical host. One challenge is that `qcow2` only allocates block addresses (via `qcow2_alloc_cluster_offset()`) when the sectors are written to disk. So `LiveMR` maintains a queue of {sector number, length} pairs (from `ext4_map_blocks()` within VMs), which are waiting to be translated. After each new data chunk is written to the `qcow2` file, the queue is examined (via `qcow2_get_cluster_offset()`) and trimmed. With a reasonable file system design, block allocations won't be too much earlier than the actual write requests – therefore the queue shouldn't grow in a uncontrolled manner. Our evaluations have included highly intensive I/O patterns (up to 16 threads doing concurrent direct or buffered I/O), and the queue of each VM never grows over 10 elements.

2 VFS File Mapping

When the composed file ("Source File" in Figure **??**) receives updated block addresses, an intuitive method to record these addresses is to directly update a source file's `inode` (`i_block[]` in ext file systems). However, this re-

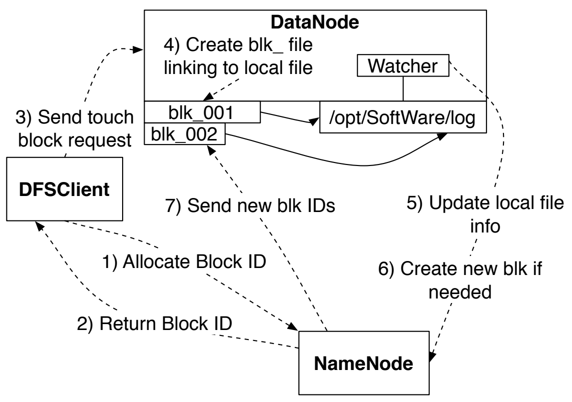
DataNode
4) Create blk_ file linking to local file
Watcher
blk_001
blk_002
/opt/SoftWare/log
3) Send touch block request
DFSClient
7) Send new blk IDs
5) Update local file info
1) Allocate Block ID
6) Create new blk if needed
2) Return Block ID
NameNode

Figure 2: I/O flow with our mechanism

quires heavy changes to the file systems and incurs moving existing array elements when one segment in the middle is expanded.

Therefore, `LiveMR` records information of each segment in the source file's extended attributes (`xattr` filed) in `ext` file systems. Each attribute has the format {destination file or device path, offset[], length[]}, where offset[] and length[] are lists representing a set of segments in the destination file. A new system call, `fsmapcreate`, is implemented for such attributes to be created, coupled with `fsmapdelete` to remove them. The current design of `LiveMR` doesn't allow writing data through the composed files. When a request arrives to read `length` bytes from the composed file at position `offset`, `LiveMR` translates it into `offset_i` and `length_i` pairs for the segments. The current implementation supports both `read` and `sendfile` system calls. To minimize the space consumption and lookup overhead, it's essential to merge segments whenever possible. In `fsmapcreate` we check if an incoming entry is adjacent to any existing ones, and whether it connects two previously disjoint ones. Our experiments show that this causes a very small overhead in read accesses. As future work, we plan to add an option to directly append incoming block addresses to the source file's `inode`. This will enable more efficient read accesses if the workload is not sensitive to the order between data chunks.

3 Programming HDFS Metadata

Finally, `LiveMR` implements the metadata protocol between local file systems and HDFS. HDSF uses DataNode local file systems to store its blocks as files prefixed by `blk_`. The basic idea is to turn those `blk_` files into links pointing to existing local files. Figure 2 illustrates the sequence of operations, and the rest of the section describes key system components.

File system shell: We create a new shell command `composeFromLocal` which takes a configuration file containing {VM: loca_path} pairs.

DFSClient: The modified DFSClient is responsible for connecting to each DataNode included in the configuration file, and creating HDFS blocks as "links" to local files. To indicate such an operation, the client sends a newly created code `TOUCH_BLOCK` together with the path of the local file.

DataNode: After receiving the `TOUCH_BLOCK` code, the DataNode creates a link to the local file. If the size of the local file is smaller than the HDFS block size (64MB by default), the link works similarly as a symbolic link. Otherwise, the link points to the middle of the local file, using the file mapping mechanism that will be introduced in section 1.1. The DataNode is then responsible for monitoring the local file and informing the NameNode of any changes. Since HDFS goes through the local file system for I/O operations (mostly through `sendfile` system call), it only needs to know the most recent size of the local file. This is done through the Linux `inotify` system calls. In particular, when the local file grows across the boundary of HDFS block size, DataNode will inform the NameNode to create a new block.

NameNode: The NameNode receives messages from DataNodes to update the size of stored blocks or to create new blocks.

Checksums and Replications: HDFS relies on content-based checksums for data integrity and replication for high availability. It is an interesting design consideration whether to keep these mechanisms for composed files – the integrity and availability should arguably be handled by the primary workloads and local file systems. In the current implementation of `LiveMR`, we have disabled checksums, and configured replication degree to be 1 for composed input files (intermediate and output files are still replicated 3 times).

3.1 Control Panel

As discussed in section **??**, users running analytics workloads are usually different from those running primary applications. `LiveMR` interacts with analytics users with a job-oriented control panel and hides Hadoop cluster details. The current implementation of the control panel is based on the OpenStack architecture. Figure 3 illustrates the design.

A user submits MapReduce jobs as compiled `jar` files to the control panel server – currently configured as the OpenStack controller for simplicity. Together with each `jar` job the user should provide a configuration file detailing each VM local file to include in the analytics in-

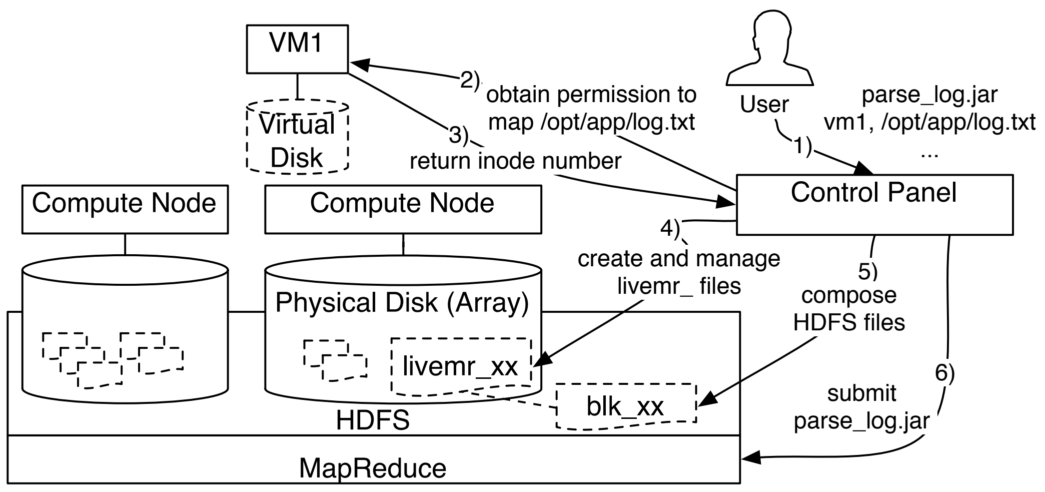

Figure 3: Control panel

put. Then the server executes the metadata protocols described in sections 1.1 2 to enforce the end-to-end mapping from VM local files to HDFS.

On each VM, a lightweight `LiveMR` script is installed to process mapping requests from the OpenStack controller and attempt to obtain permission from the file owner to map the file to the hypervisor and eventually to HDFS. The current design relies on messages (e.g., Linux `mail`) to send permission requests to file owners, who grants permissions by setting a special `xattr` on the mapped files. The `ext4_map_blocks()` function will recognize this `xattr` and send block device `ioctl` and eventually `SCSI` commands to `QEMU`. If the permission is granted, the `inode` number of the mapped file is acknowledged back to the OpenStack controller. If the analytics user has already obtained coarse-grained file access permissions (e.g., through `ssh` keys), the protocol can be omitted and the `LiveMR` script will directly set the special `xattr`.

On each compute node, the control panel composes a set of local files by catching block addresses passed down from `QEMU`, and manages them as a middle ground between VM local files and HDFS. Those files are "stored" at a designated path (without occupying storage space) with a common prefix (`livemr_`). Technically it is possible to directly compose HDFS `blk_` files to point to segments in the VM image files. However, `LiveMR` uses `livemr_` files to allow a decoupled and extensible architecture. On one hand, the `livemr_` files appear as regular files on the compute nodes and are therefore compatible with other analytics applications. On the other hand, this allows `LiveMR` to communicate with HDFS via the designated `DFSClient` interface, instead of requiring HDFS to expose its internal block names to external configuration files. Because the current implementation of VFS file mapping supports a maximum of 64 contiguous segments (section 1.1), `LiveMR` creates a new `livemr_` file when the current one reaches this limit. The new `livemr_` file name is then added to the HDFS file via the `composeFromLocal` command introduced in section 2.