Adaptive File Management for Scientific Workflows on the Azure Cloud

Radu Tudoran[‡], Alexandru Costan[*], Ramin Rezai Rad[†], Goetz Brasche[†] and Gabriel Antoniu[*]
[‡]Microsoft Research-Inria Joint Centre, Palaiseau, France,
{radu.tudoran}@inria.fr
[*]Inria Rennes-Bretagne Atlantique, Rennes, France,
{alexandru.costan, gabriel.antoniu}@inria.fr
[†]Microsoft Research ATLE, Aachen, Germany,
{ramrezai, goetz.brasche}@microsoft.com

***Abstract*—Scientific workflows typically communicate data between tasks using files. Currently, on public clouds, this is achieved by using the cloud storage services, which are unable to exploit the workflow semantics and are subject to low throughput and high latencies. To overcome these limitations, we propose an alternative leveraging data locality through direct file transfers between the compute nodes. We rely on the observation that workflows generate a set of common data access patterns that our solution exploits in conjunction with context information to self-adapt, choose the most adequate transfer protocol and expose the data layout within the virtual machines to the workflow engines. This file management system was integrated within the Microsoft Generic Worker workflow engine and was validated using synthetic benchmarks and a real-life application on the Azure cloud. The results show it can bring significant performance gains: up to 5x file transfer speedup compared to solutions based on standard cloud storage and over 25% application timespan reduction compared to Hadoop on Azure.**

I. Introduction

A large class of scientific applications can be expressed as workflows, which describe the relationship between individual computational tasks and their input and output data in a declarative way. To achieve rapid turnaround, workflows require adequate infrastructures, like clouds, for a proper execution. One missing link that limits a larger adoption of these infrastructures is the data management, as clouds mainly target web and business applications, and lack specific support for data-intensive scientific workflows. Typically, a workflow consists of a set of loosely-coupled tasks linked via data- and control-flow dependencies. Unlike tightly-coupled applications (e.g. MPI-based) communicating directly via the network, workflow tasks exchange data through (large) files.

Currently, the workflow data handling in the clouds is achieved using either some application specific overlays that map the output of one task to the input of another in a pipeline fashion, or, more recently, leveraging the MapReduce programming model (e.g. Amazon Elastic MapReduce [1], Hadoop on Azure - HDInsight [2]). However, most scientific applications don't fit this model and require a more general data orchestration, independent of any programming model. Research into extensions of MapReduce attempt to bridge these differences, taking inspiration from MPI, but they are limited to iterative [3], [4] or distributed [5] versions that don't exploit the workflow semantics. As each individual task runs on a separate virtual machine (VM), workflows need a high performance storage system that would allow VMs to concurrently access shared data. However, today's reference commercial clouds only provide object stores such as S3 or Azure Blobs accessed through high-latency HTTP interfaces, that also require to change the way data is managed in order to adapt to the actual access method (files vs. objects) [6]. An alternative would be to deploy a parallel file system in the cloud in order to exploit data locality. Nevertheless, most file systems need special configuration or handling to get them to work in a virtualized environment, while others cannot be executed at all, since they require kernel modifications not allowed by most cloud providers [7], [8].

In this paper, we specifically address these issues. We introduce an approach for the efficient sharing and transfer of input/output files between the compute instances in the cloud. We advocate storing data on the compute nodes and transferring files between them directly, in order to exploit data locality and to avoid the overhead of interacting with a shared file system. Under these circumstances, we propose a file management service that enables high throughput through multiple transfer strategies (e.g. FTP-based, BitTorrent-based, etc.). These strategies are implemented as dynamically loadable and easily extensible modules. Our proposal does not require any foreknowledge of the access pattern and dynamically adapts to the workflow context (data size, format, access, resource cost) by choosing the most efficient transfer protocol. This work is integrated within the Generic Worker (GW [9]) workflow engine designed by Microsoft Research. We summarize our contributions as follows:

- We present an overview of the workflow data management issues on clouds (Section II-C).
- We introduce an approach that optimizes the workflow data transfers on clouds by means of adaptive switching between several inter-VM file transfer protocols using context information (Sections III and IV).
- We propose an implementation of these design principles in a file management system integrated within Microsoft's GW workflow engine (Section V).
- We experimentally evaluate the benefits of our approach on hundred of cores of the Windows Azure cloud in three different contexts: synthetic bench-

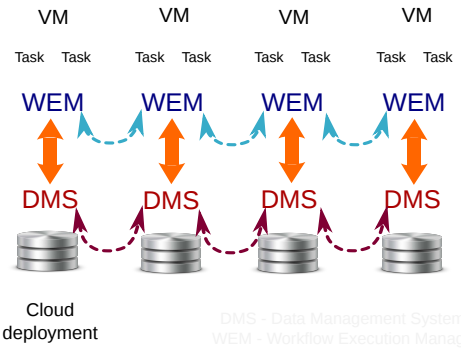

Fig. 1. A typical cloud deployment containing a workflow engine and a data management system which collaborate in order to optimize the data processing by migrating and scheduling the data or the tasks

marks reproducing scientific data patterns, a real-life application from bio-informatics and a MapReduce processing (Section VI).

II. Context and Related Work

This section presents the necessary background and elaborates on our observations that help motivate the cloud-based workflow data management research.

A. The Need of Adaptive File Management for Workflow Processing on Clouds

Executing a scientific workflow in the cloud involves moving its tasks and files to the execution nodes [10], [11]. This data movement is critical for performance and costs since when a task is assigned to an execution node, some of its required files may not be available locally. With the file sizes handled by scientific workflows increasing to petabyte levels, data handling becomes a bottleneck, impacting on the workflow's makespan and costs. Thus, properly scheduling the tasks according to the data layout within the compute VMs or placing the data according to the computation pattern becomes a necessity. In order to achieve this, a two-way communication between the workflow engine and the data management system is required, as illustrated in Figure 1.

B. Existing Data Management Solutions

Traditional techniques commonly found in scientific computing (e.g. relying on parallel file systems) are not always adequate for processing big data on clouds. Such architectures usually assume high-performance communication between computation nodes and storage nodes (e.g. PVFS [13], Sector [12]). This assumption does not hold in current cloud architectures, which exhibit much higher latencies between compute resources and storage resources. Although a large number of network storage systems exist, few of them can be deployed on clouds. Unlike our approach, most of them handle data transfers statically and need special configuration or handling to get them to work in a virtualized environment, while others cannot be executed at all, since they require kernel modifications (e.g. Lustre [14]), which are not allowed by most cloud providers within the limited user permissions in which the cloud leased resource are used.

Several workflow management systems were specifically proposed for scientific applications running on clouds. e-Science Central [10] enables non-programmer scientists to harness vast amounts of storage and compute power for running applications in batches without user interaction. Dryad [15] and Hadoop [16] allow for the distributed processing of large data sets across clusters of computers. However, they all force scientists to adopt rigid programming models (e.g. MapReduce, DAG based flows) and typically rely on high-latency HTTP protocols, TCP pipes or shared memories for static data transfers. In contrast, our approach is independent of any programming interface and dynamically adapts to the execution context. Pegasus [17] relies on a peer-to-peer file manager similar to ours, when deployed on Amazon EC2. However, they use static transfers between the VMs and their approach is outperformed by a shared file-system.

Cloud based services, like Amazon's CloudFront [18], use the geographical distribution of data to reduce latencies of data transfers. Similarly, [19] and [20] considered the problem of scheduling data-intensive workflows in clouds assuming that files are replicated in multiple execution sites. These approaches can reduce the makespan of the workflows but come at the cost and overhead of replication, which is considerable for large datasets. In contrast, we don't use multiple copies of data, but rather exploit the data access patterns to allow per file optimizations of transfers. Finally, several studies have investigated the performance of data management for real science applications on clouds [21] [22], but with few exceptions they have focused on tightly-coupled MPI applications or loosely-coupled MapReduce ones. Our work targets scientific workflows, with more general data interactions, non restricted to a specific programming model.

C. File Transfer Support for Workflow Management

Requirements for cloud-based workflows. In order to support data-intensive workflows, a cloud-based solution needs to: 1) Adapt the workflows to the cloud environment and exploit its specificities (e.g. running in user's virtualized space, commodity compute nodes, ephemeral local storage); 2) Optimize data transfers to provide a reasonable time to solution; 3) Manage data so that it can be efficiently placed and accessed during the execution.

Data management challenges. *Data transfers* are affected by the instability and the heterogeneity of the cloud network. There are numerous options, some providing additional security guarantees (e.g. TLS) others designed for a high throughput (e.g. BitTorrent). *Data locality* aims to minimize the amount of data movement and to improve end-application performance and scalability. Addressing data and computational problems separately forces much data movement which will not scale to tomorrow's exascale datasets and millions of nodes, and will yield significant underutilization of the resources. *Metadata management* plays an important role as, typically, the data relevant for a certain task can be stored in multiple locations. Logical catalogs are one approach to providing consistent information about the location of the data items.

Programming challenges. So far, *MapReduce* has been the "de-facto" cloud computing model, complemented by a

number of variations of languages for task specification [23], [15]. They provide some data flow support, but all require a shift of the application logic into the MapReduce model. Workflow semantics go beyond the map-shuffle-reduce-merge operations, and deal with data placement, sharing, inter-site data transfers etc. Independently of the programming model of choice, be it MapReduce, scripting languages, process descriptions like BPEL or third party APIs, they all need to address the following issues: support large-scale parallelism to maximize throughput under high concurrency, enable data partitioning to reduce latencies and handle the mapping from input, intermediate and output data to cloud logical structures to efficiently exploit its resources.

Targeted workflow semantics. In order to address these challenges we studied 27 real-life applications [24] from several domains (bio-informatics, business logic, simulations etc.). We identified a set of core-characteristics shared by a vast majority of them, which impact on the data management:

- The common *data patterns* are: broadcast, pipeline, gather, reduce, scatter.
- Workflows are composed of batch jobs with well-defined *data passing schemas*. The workflow engines execute the batch jobs (e.g. take the executable to a VM instance; bring the needed data files and libraries; run the job and retrieve the final result) and perform the required data passing between jobs.
- The input and the output of the batch jobs are files, usually *written once*.
- The batch jobs and their inputs and outputs can be *uniquely identified*.

Our focus is on how to efficiently handle and transfer workflow data between the VMs in a cloud deployment. We argue that keeping data in the local disks of the VMs is a good option considering that for such workflows, most of the data files are usually temporary - they must exist only to be passed from the job that produced it to the one it will further process it. With our approach, the files from the virtual disk of each compute node are made available to all other nodes within the deployment. Caching the data where it is produced and transferring it directly where it is needed reduces the time for data manipulation and minimizes the workflow makespan.

III. Our Approach: an Adaptive File Management System

A. Design overview

Our proposal relies on four key design principles:

Exploiting the data locality. The cumulative storage capacity of the VMs leased in one's deployment easily reaches the TBs order. Although batch jobs store their input and output files on the local virtual disks, most of the storage capacity remains unused. Meanwhile, workflows typically use remote cloud storage (e.g. Amazon S3, Azure Blobs) for sharing data across compute VM instances [10], [9]. This is costly and highly inefficient, due to high latencies, especially for temporary files that don't require persistent storage. Instead, we propose aggregating parts of the virtual disks in a shared common pool, managed in a distributed fashion, in order to share data between the computing nodes directly.

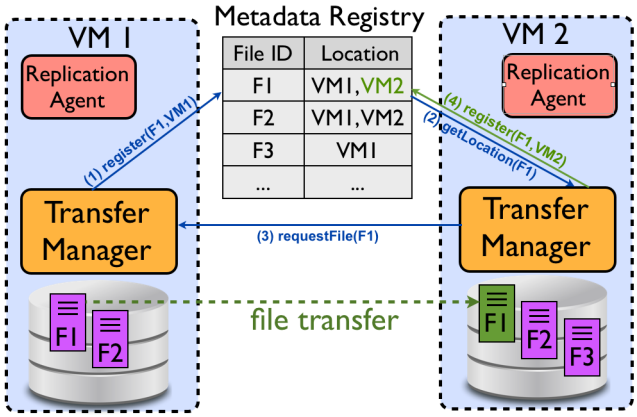

Fig. 2. Architecture of the File Management System. Operations for transferring files between VMs: upload (1), download (2,3,4).

Storage hierarchy. We advocate the use of a hierarchy for workflow data handling in the cloud, comprising several layers: in-memory storage at the top, local disks, shared VM-based storage and finally the remote cloud storage at the bottom. As memory and storage devices move down the hierarchy they reduce in performance but tend to rise in capacity and costs. As opposed to a classical computer architecture, the costs tend to increase towards the base of the hierarchy as the remote storage comes at an extra-cost while the local resources are available for free, from a storage perspective (one has to pay only for the compute cycles). Files are moved up and down the storage hierarchy via staging and migration operations, respectively, based upon data access patterns, resource availability and user requests.

Integrate multiple transfer methodologies. Adopting several ways to perform the file transfers, like peer-to-peer, direct or parallel transfers, and dynamically choosing between them at runtime based on context information allows to exploit workflow specific data access patterns. This also opens the avenue for customization: users are able to easily add transfer modules by implementing a simple API and by providing the transfer method of their choice, able to leverage the application semantics.

No modification of the cloud middleware. Data processing in public clouds is done at user level, which restricts the application permissions to the virtualized space. Our solution is suitable for both public and private clouds, as no additional privileges are required.

B. Architecture

The simplified schema of a distributed architecture that integrates our approach is depicted in Figure 2. The input and output files of the workflow batch tasks are stored on the local disks of the VMs. We introduce three components that enable sharing these files across the compute instances:

The Metadata Registry holds the file locations (i.e. maps files to VMs). The metadata is organized as key-value pairs: *file ids* (e.g. name, user, sharing group etc.) and *locations* (the information required by the transfer module to retrieve the file). It is organized as an all-in-memory distributed hash-table

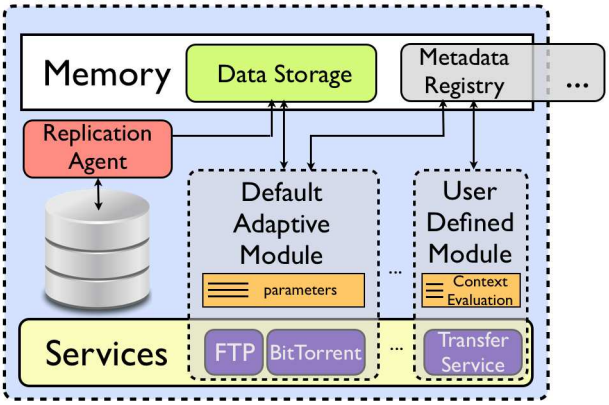

Fig. 3. The File Management System components within a Virtual Machine.

that holds a large number of small items (the key-value pairs) which are accessed by the VMs. In a general usage scenario, a concurrency handling mechanism for writes/updates would be needed for the Metadata Registry, usually provided at the expense of performance. However, as discussed in Section II-C, in the context of file management for workflow execution, the files are produced and written by a single task and uniquely identified: there are no situations in which two tasks request the Metadata Registry to publish the same new file. Therefore, there is no need for strong verification mechanisms (e.g. global locks, global queries) to detect and solve eventual duplication conflicts.

The Transfer Manager is started as a service on each VM. Applications (e.g. workflow engines) interact with the local Transfer Managers through a simple API to perform file *uploads* and *downloads*. The upload operation is equivalent to sharing a file: the Transfer Manager will advertise it by creating a record in the Metadata Registry. Hence, the cost of uploading a file is $O(1)$ (independent of the data size) and it represents the time to write the metadata. The download operation has two phases: retrieving the file information from the Metadata Registry and fetching the data from the VM holding it via the Transfer Manager on source VM. The main goal here is to minimize the number of read and write operations that need to be performed for passing a file from a task to another. As seen in Figure 3, only one read and one write operation are needed for such a direct transfer. Taking into consideration that for transferring the files, multiple options are available, this component is designed in a modular fashion such that it is easy to plug in different transfer back-ends (i.e. libraries corresponding to a data transfer technology), among which the solution most appropriate for a particular context is selected. Figure 3 presents the components of the Transfer Manager present in a VM. Essentially, the system is composed of user deployed or default transfer modules and their service counter parts, available on each compute instance. Clients (e.g. workflow engines) interact with the Transfer Manager to download and upload files, via a cloud based API.

The Replication Agent is an optional component designed to balance the load of multiple accesses to a file through replication. As such, it manages several replication strategies and policies within the file management system. The agent works as a service that is started on each VM and communicates via a queue-based messaging system. In addition to the orchestration and the replication of files, the component has the role of evaluating the gains brought by the replicas in each transfer context. These gains (i.e. the transfer time reduction) are correlated with a storage cost schema that we propose (see Section IV), to dynamically determine the appropriate number of replicas for each context. The system can be further extended to provide multiple replication schemes as discussed in [25], [26], by scheduling the replica placement in collaboration with the workflow engine.

Algorithm 1 The context-based transfer module selection

```
1: procedure AdaptiveTransferModuleSelection
2:     JobDescription = Client.getJobDescription()
3:     for all module in UserDeployedModules do
4:         score = module.ContextEvaluation(JobDescription);
5:         best_score = max(best_score, score);
6:     end for
7:     if best_score > user_defined_threshold then
8:         TransferModule = BestModule;
9:     else
10:        TransferModule = DefaultAdaptiveModule();
11:    end if
12:    Client.notify(TransferModule, JobDescription);
13:    return TransferModule;
14: end procedure
```

IV. Zoom on the Adaptive Workflow File Transfers

Users can deploy their own transfer modules by means of a straightforward API. This requires to extend the transfer protocol with an evaluation function for scoring the context. The score is computed by aggregating a set of weighted context parameters (e.g. number or size of files, replica count, resource load, data format etc.), where the weights reflect the relevance of the current transfer module for each specific parameter (e.g. a fast memory-to-memory data transfer protocol will favor transfers of many small files through higher weights for these parameters). The module with the best score is chosen for each transfer, as shown in Algorithm 1. If no user modules are deployed or none of them fits the workflow context, a *Default Adaptive Module* that we provide is chosen.

The default module selection strategy that we provide is presented in Algorithm 2. This uses a set of parameters defined by users in an XML file (e.g. size limits of files to be fitted in memory, replicas count, etc.). The selection is done based on weights assigned to these parameters, that define the transfer context. The importance of each parameter is rated by both clients (e.g. workflow engines) and the Replication Agent. The latter can in fact modify the transfer context by increasing the number of replicas if the transfer speedup obtained comes at a cost that fits the budget constraints. Currently, the selection is done between 3 transfer protocols that we provide within our framework:

- **In-Memory:** for small files or for situations in which the leased infrastructure has enough memory capacity, keeping data in the memory across VMs becomes interesting. This provides one of the fastest methods

Algorithm 2 The default parameterized module for a weighted transfer selection

```
1: procedure DEFAULTADAPTIVEMODULE
2:     ReadAdaptiveParameters(UserDefinedParams_XML)
       ▷ get the weights recommended by the client and by the
       Replication Agent based on the budget
3:     (cl_size_weight, cl_replica_weight) =
       Client.getRecomandedScore();
4:     (re_size_weight, re_replica_weight) =
       Replicator.getRecomandedScore(user_def_cost_ratio);
5:     ▷ try to speedup the transfer based on replicas within the
       budget constraints
6:     Replicator.updateContextForSpeedup(
       JobDescription, user_def_cost_ratio_threshold)
7:     ▷ evaluate the transfer context based on file size and
       replicas count
8:     if JobDescription.Output.Size × (cl_size_weight
       +re_size_weight)< user_def_memory_threshold then
9:         return InMemoryModule;
10:    else
11:        if JobDescription.Replicas ×(cl_replica_weight
           +re_replica_weight) < user_def_replica_threshold then
12:            return TorrentModule;
13:        else return DirectLinkModule;
14:        end if
15:    end if
16: end procedure
```

to handle it, boosting the performance especially for scatter and gather/reduce data access patterns. A percentage of the VM memory will be dedicated to the shared memory system, which can reach an aggregated capacity in the order of GBs, from all the nodes.

- **FTP:** for large files, that need to be transferred from one machine to another, direct TCP-transfers are privileged. FTP seems a natural choice for interoperability reasons. The data access patterns that benefit most from this approach are pipeline and gather/reduce.
- **BitTorrent:** for data patterns like broadcast, multicast or scatter, having multiple replicas enables a throughput increase and a load balance as clients collaborate in retrieving the data. Thus, for scenarios involving replication (above a user defined threshold) of large datasets, we rely on BitTorrent. These gains in performance can be enhanced for time critical transfers or for files that are highly accessed by increasing the number of replicas at the expense of the extra storage space that the replicas occupy.

Discussion: the cost of data dissemination. As replication has a direct impact on the transfer performance, we propose a cost model that gives hints on how to adapt the replicas count with respect to transfer speedup. We start by associating a cost for storing data on a VM disk. Although there is no additional cost for the local virtual disks, the storage capacity is limited to several hundreds GBs depending on the VM type. We define this cost as the ratio between the capacity of the VM disk and its cost: $cost_{MB} = \frac{VM\_Pricing}{Local\_Disk\_Capacity}$. Then, the cost of having N_R replicas, each having $Size$ MB, is $cost_{Replicas} = N_R * Size * cost_{MB}$. Next, we examine the time gain obtained from each replica. We assume a linear function starting from one replica, which implies a transfer time of $time_{transfer} = \frac{Size}{Throughput}$, down to a transfer time of 0, when there is a replica on all nodes ($N_R = N_{Nodes}$). This leads to the next function defining the gained time:

$$time_{gain} = \frac{Size}{Throughput} * (1 - \frac{N_{Nodes} - N_R}{N_{Nodes} - 1}),$$

varying from 0 for one replica up to $time_{transfer}$, when data is already present and no transfer is needed. Hence, we are able to associate a cost for speeding-up the data dissemination as the ratio $\frac{time_{gain}}{cost_{Replicas}}$. The *Default Adaptive Module* integrates this model within the Replication Agent, using a distributed, queue-based, replication service, in order to dynamically adjust the number of replicas. Users can define a certain cost that they are willing to pay for speeding the transfer, as a ratio parameter (i.e. $user\_def\_cost\_ratio\_threshold$). Based on it and on the transfer parameters (i.e. size, transfer throughput), the Replication Agent can scale the replicas in the system and choose one transfer module over another in order to decrease the transfer time within the cost boundaries imposed by the extra storage space used, as shown in Algorithm 2.

V. IMPLEMENTATION

We implemented a prototype of the file management system and integrated it within the Microsoft Generic Worker workflow engine, by replacing its default data storage backend, which relied on Azure Blobs. Using the provided API, the information about the tasks' files is passed from the GW task manager to our system. As future work, the task scheduler of the GW will be extended, in order to use the data locality information that our solution provides. The GW engine was chosen, as it facilitates the process of porting existing science applications to clouds (in particular, Azure) by deploying and invoking them with minimal effort and predictable cost-effective performance. While the system is optimized for Azure, it can be easily extended to other cloud technologies. The GW supports this generalization through a set of pluggable components with standardized interfaces that allow a simple migration to other clouds, only by replacing the runtime component.

The Metadata Registry has to efficiently support a large set of small key-value pairs. The service must be distributed and reachable from all the node instances. Several implementation alternatives cope with these requirements: in-memory databases, Azure Tables, Azure Caching [27]. We chose the latter as it easily allows a percentage of the VM's memory to be dedicated to caching. Depending on the leased resources, the VM memory can reach capacities in the order of GBs, making it viable for data sharing. Our preliminary evaluations show that the Azure Caching delivers better performances than the Azure Tables (10 times faster for small items) and has a low CPU consumption (unlike a database).

The Transfer Manager needs to support the seamless integration of new, user defined, transfer methods. To achieve this, we opted for the Management Extensibility Framework [28], which allows the creation of lightweight extensible applications. The default *in-memory module* is based also on the Azure Caching system. The *FTP module* relies on an open source library [29], tuned to harness the cloud specificities:

as a deployment is virtually isolated, authentication between the nodes is redundant, so it was removed; the chunk size of data read/written to the TCP connection was increased to 1 MB for a higher throughput. The *BitTorrent module* uses MonoTorrent [30], an open source protocol implementation. In a typical scenario with multiple peers sharing data across wide area networks, the default packet size is set to 16 KB. However, our initial experiments showed that this chunk size is too small for transfers between nodes within a data center. After experimenting with varying chunk sizes (up to 16 MB), we chose a 1 MB chunk, which increases the throughput up to 5 times. For distributing the load, trackers are started within each VM, at which the peers (i.e. the Transfer Managers using the torrent module) that seed files register, while the ones that download inquire for seeder discovery.

The Replication Agent is implemented as a service that runs as a background process in the cloud VMs and provides two functions. On the one hand, each agent acts as a worker polling for jobs, that specify the files to replicate. For job communication between agents, a message passing schema over the Azure Queue was built. On the other hand, the Replication Agent implements the space/cost/performance strategy described in Section IV to create the appropriate number of replicas. This is done by creating replication jobs that are assigned to the other agents in the system, which transfer the data when the network bandwidth of the corresponding VM is not saturated by the Transfer Manager.

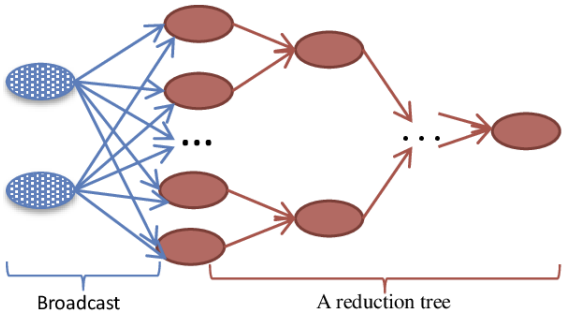

Fig. 4. Workflow schema for the synthetic benchmark.

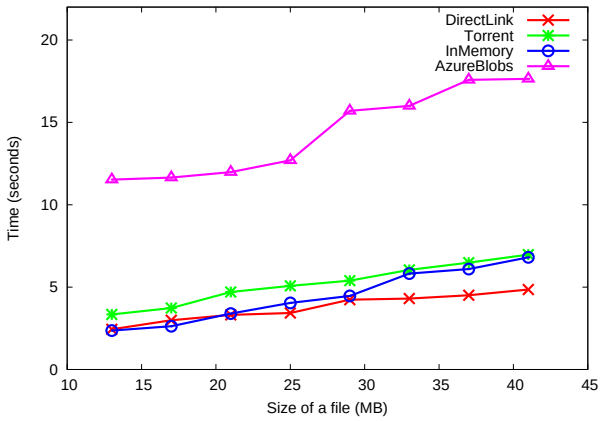

Fig. 5. Average transfer time per job when 2 input files are downloaded and one is uploaded

VI. Evaluation

In this section we evaluate our approach both in synthetic and real-life settings. Our experimental setup consists of 50 Medium nodes (2 CPU cores, 3.5 GB memory, 490GB local disk) from the Azure cloud deployed in Europe and US data centers. We analyze the file transfer times of our default modules and their impact on the makespan of a real-life application, compared to the usage of a shared cloud storage service (Azure Blobs [6]) and a MapReduce framework (HDInsight - Hadoop on Azure[2]).

A. Synthetic benchmarks

Our first series of experiments analyze the adaptive parameters of our solution in order to better understand its behavior and the different trade-offs involved. To this end, we implemented a simple benchmarking workflow (Figure 4) that encapsulates two data access patterns (broadcast and pipeline within a reduction tree). The workflow is composed of 38 distributed identical jobs, with 20 of them on the first tree layer. Each job takes 2 input files containing numbers, applies several mathematical operations and then stores the result in an output file, used by the tasks on the next layers. Additionally, 2 other jobs (the ones in the left in Figure 4) are used for staging-in the initial input files.

Scenario 1: Small files, no replication. This scenario is useful to determine the threshold up to which the in-memory transfer is efficient for cloud VMs. Figure 5 presents the average time of a job manipulating 2 input and one output file with different sharing solutions. Not surprisingly, managing the files inside the deployment reduces the transfer times up to a factor of 4, compared to the remote shared cloud storage (Azure Blobs). We zoom on the behaviour of our transfer modules by eliminating the time series for the Azure Blobs in Figures 6 and 7, which depict the transfer times for downloading / uploading files for the jobs of the synthetic workflow. We notice that for small files, the in-memory solution delivers the best results. With increasing sizes, transferring files directly becomes more efficient, as the in-memory module must handle more fragments - the files are fragmented/defragmented into 4 MB chunks (the maximum size of an object in Azure Cache). Additionally, retrieving more fragments leads to increasing performance variations, which should be avoided as scientific applications require predictable performance [31]. Based on these, we retain a threshold of 15 MB for the size of the files shared in-memory. The torrent module pays the price of an extra operation for hashing the file and generating the ".torrent" metadata, used by peers for download, which makes it inefficient for small non-replicated files. We note that the upload time is $O(1)$, as discussed in Section III-B, involving only the registration of the file to the Metadata Registry. However, in memory-based write operations are $O(n)$ as all fragments must be written in the memory, giving another argument for bounding the maximum file size of such transfers.

Scenario 2: Medium to large files, replication enabled. Next, we evaluate the impact of our approach in an unfavorable scenario: sharing large files, replicated across the cloud infrastructure. The stage-in jobs (the ones in the left of Figure 4) of our synthetic workflow generate 5 replicas for their output data; in Azure Blobs the number of replicas is automatically set to 3 and the files are transparently distributed within the storage service. We notice from Figure 8 that our adaptive solution (dynamically switching between transfer methods) achieves

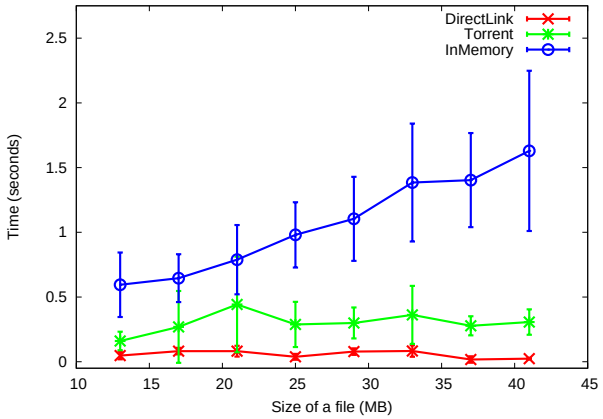

Fig. 6. Upload time for a file

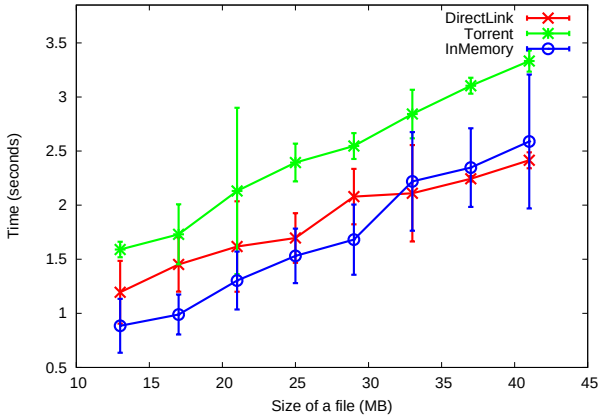

Fig. 7. Download time for 2 files

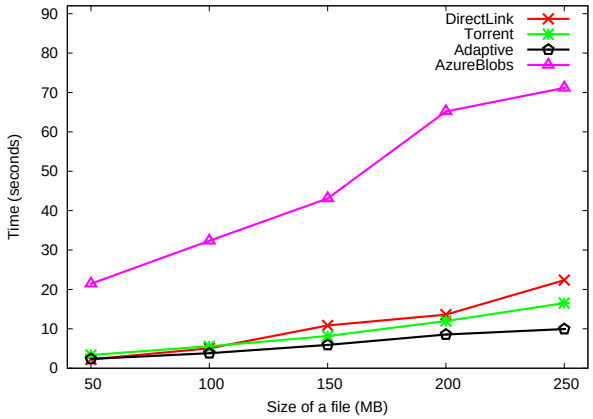

Fig. 8. Transfer time per job when 2 input files are downloaded and 1 is uploaded

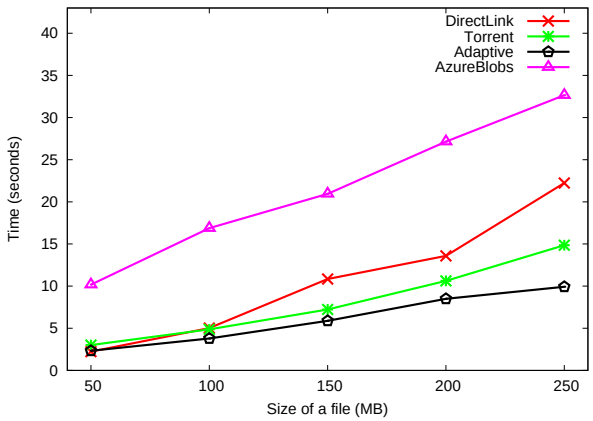

Fig. 9. Download time for the 2 input files, when increasing the file size

a 2x speedup compared to a static file handling (within the deployment) and a 5x speedup compared to Azure Blobs (remote storage). As the upload time is almost constant (less than 1 second) for our system, we depict in Figure 9 the time to download the 2 input files, when increasing their sizes. With multiple seeders for the replicas, the torrent-based module is more efficient. In the broadcast phase of the workflow, torrents perform better, while in the reduction phase, direct link will work better for the pipeline transfers. Our adaptive solution exploits these patterns and switches in real-time between these two modules.

Scenario 3: MapReduce processing, Word Count. This third set of experiments considers a typical MapReduce processing in order to assess whether a general purpose workflow engine enhanced with our file management scheme can meet the performance of a dedicated MapReduce engine. Figure 10 presents the average time of the mappers and reducers with Hadoop on Azure (HoA), the default Generic Worker using AzureBlobs (GW) and the extended version relying on our adaptive file manager (GW++). The input data size is 650 MB, processed in two scenarios: with 20 mappers (32 MB per job) and 40 mappers (16 MB per job); the number of reducers is fixed at 3. We notice that GW++ achieves a 25% speedup compared to Hadoop, building on its access pattern-awareness. In the next experiment (Figure 11), we increased the input data size up to 50 GB and fixed the number of mappers (100) and reducers (5). The gain in time for the GW++, which reaches 10% as the input dataset is increased, is due to the adaptive behavior of the mapper, which switches between direct link and in-memory transfers, as opposed to Hadoop which always uses HDFS disk-based data sharing. In fact, HDFS performs the data manipulation locally, within the compute nodes, which implies that at least half of the gain comes from the adaptive behavior, while the remaining would come from replacing HDFS.

B. *Case study: BLAST - a bio-informatics application*

Our next series of experiments focuses on real-life scientific applications. We illustrate the benefits of our approach for one such application: BLAST, a workflow for comparing primary biological sequences to identify those that resemble above a certain threshold. BLAST is representative of a large class of scientific workflows that split their initial data into sub-domains which are analysed in parallel, iteratively. The BLAST workflow is composed of 3 types of batch jobs. A *splitter* partitions the input file (up to 800MB in our experiments) and disseminates it to the set of distributed BLAST jobs. The core algorithm (the *BLAST jobs*) matches the input file with reference values stored in 3 database files (the same for all the jobs). Finally, the *assembler* job aggregates the result from the BLAST jobs into a final result.

Figure 12 presents the makespan of the BLAST analysis and Figure 13 shows the average file upload and download times for a BLAST job, when their number is increased. The experiments were carried out using the Generic Worker with Azure Blobs and our adaptive solution. Increasing the number

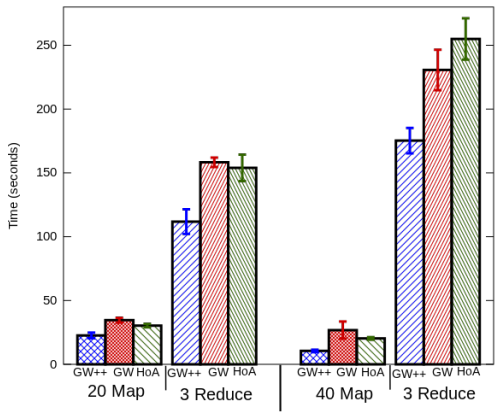

Fig. 10. Map and Reduce times for WordCount (32 MB vs. 16 MB / map job).

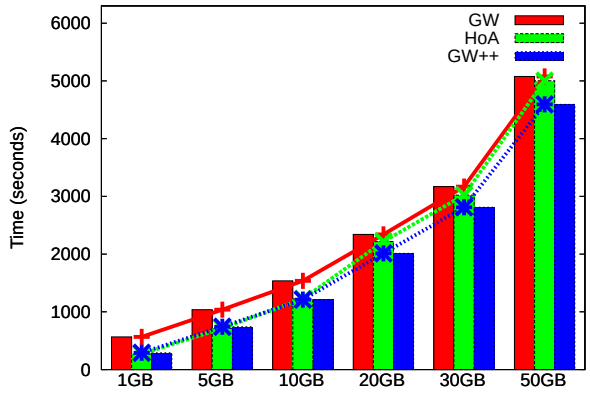

Fig. 11. WordCount execution time for large datasets and fixed mappers/reducers.

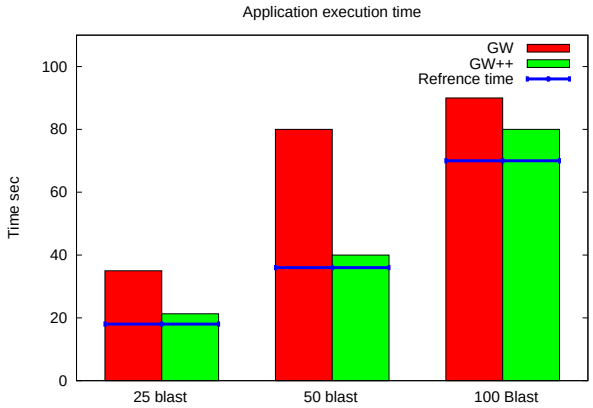

Fig. 12. The BLAST workflow makespan: the compute time is the same (marked by the horizontal line), the remaining time is the data handling

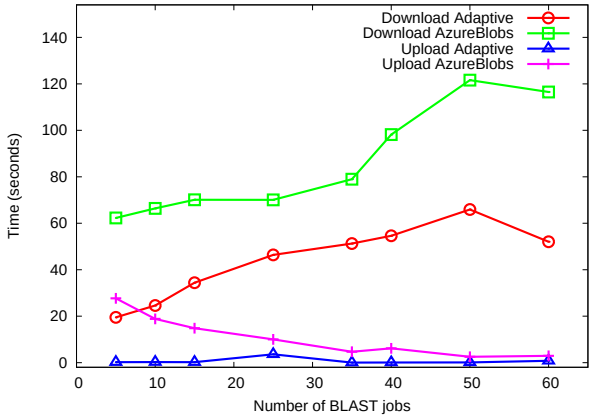

Fig. 13. Average times for staging data in and out for a Blast job when the number of jobs is increased

of jobs results in smaller temporary files (the size of the database files to be broadcasted remains the same, approx. 1.6 GB), with a higher access concurrency. As the number of nodes available for the experiment was fixed (50), the tasks are executed in waves when their number exceeds the available VMs. This explains the drop in the average transfer time when running more than 50 BLAST jobs (Figure 13). We notice that by adapting the transfer method to the data access pattern, the computation time is significantly reduced. When we isolate the file handling times, we observe that these are reduced to half per job with the adaptive solution.

VII. Conclusions

Currently, clouds rely on network overlays and MapReduce style processing for workflow data handling. This paper proposes a cloud-based alternative: a file management service, integrated within Microsoft's Generic Worker, that enables efficient file transfers directly between the compute nodes. Our approach is highly adaptive and can choose between several dynamically loadable transfer modules the most suited for a specific data movement. It does not require any past trace of the transfer, but rather relies on context data (resource status, data size and format etc.) and on the detected access pattern. This solution brings a transfer speed-up of up to a factor of 5 compared to using the default cloud storage, while the execution time of the applications is reduced with 25 % when compared to Hadoop on Azure.

Thanks to these encouraging results, we plan to further investigate the potential benefits of exploiting the description of the entire workload, if available, when scheduling the file transfers. In particular, we see a good potential to reduce the transfer decision overhead by means of predictions and plan to investigate this issue more closely. Furthermore, an interesting direction to explore is the closer integration between the workflow engine and the file system deployed on the compute nodes, in order to exploit data placement hints for file transfers.

Acknowledgment

This work was supported by the joint INRIA - Microsoft Research Center. The experiments presented in this paper were carried out using the Azure Cloud infrastructure provided by Microsoft in the framework of the A-Brain project. The authors would like to thank the Azure support teams from EMIC for their valuable input and feedback.

References

[1] "Amazon elastic mapreduce," http://aws.amazon.com/elasticmapreduce/.

[2] "Hdinsight (hadoop on azure)," https://www.hadooponazure.com/.

[3] T. Gunarathne, B. Zhang, T.-L. Wu, and J. Qiu, "Scalable parallel computing on clouds using twister4azure iterative mapreduce," *Future Gener. Comput. Syst.*, vol. 29, no. 4, pp. 1035–1048, Jun. 2013.

[4] R. Tudoran, A. Costan, and G. Antoniu, "Mapiterativereduce: a framework for reduction-intensive data processing on azure clouds," in *Proceedings of third international workshop on MapReduce and its Applications Date*, ser. MapReduce '12. New York, NY, USA: ACM, 2012, pp. 9–16.

[5] L. Wang, J. Tao, H. Marten, A. Streit, S. U. Khan, J. Kolodziej, and D. Chen, "Mapreduce across distributed clusters for data-intensive applications," in *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, ser. IPDPSW '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 2004–2011.

[6] B. Calder and et al, "Windows azure storage: a highly available cloud storage service with strong consistency," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, ser. SOSP '11. New York, NY, USA: ACM, 2011, pp. 143–157.

[7] G. Juve, E. Deelman, K. Vahi, G. Mehta, and B. Berriman, "Scientific workflow applications on amazon ec2," in *In Cloud Computing Workshop in Conjunction with e-Science*. IEEE, 2009.

[8] G. Juve, E. Deelman, G. B. Berriman, B. P. Berman, and P. Maechling, "An evaluation of the cost and performance of scientific workflows on amazon ec2," *J. Grid Comput.*, vol. 10, no. 1, pp. 5–21, Mar. 2012.

[9] Y. Simmhan, C. van Ingen, G. Subramanian, and J. Li, "Bridging the gap between desktop and the cloud for escience applications," in *Proceedings of the 2010 IEEE 3rd International Conference on Cloud Computing*, ser. CLOUD '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 474–481.

[10] H.Hiden, S. Woodman, P.Watson, and J.Caa, "Developing cloud applications using the e-science central platform." in *Proceedings of Royal Society A*, 2012.

[11] K. R. Jackson, L. Ramakrishnan, K. J. Runge, and R. C. Thomas, "Seeking supernovae in the clouds: a performance study," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, ser. HPDC '10. New York, NY, USA: ACM, 2010, pp. 421–429.

[12] R. L. Grossman, Y. Gu, M. Sabala, and W. Zhang, "Compute and storage clouds using wide area high performance networks," *Future Gener. Comput. Syst.*, vol. 25, no. 2, pp. 179–183, Feb. 2009.

[13] P. H. Carns, W. B. Ligon, III, R. B. Ross, and R. Thakur, "Pvfs: a parallel file system for linux clusters," in *Proceedings of the 4th annual Linux Showcase & Conference - Volume 4*, ser. ALS'00. Berkeley, CA, USA: USENIX Association, 2000, pp. 28–28.

[14] P. Schwan, "Lustre: Building a file system for 1000-node clusters," in *Linux Symposium 2003*.

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

[16] "Hadoop," http://hadoop.apache.org/.

[17] R. Agarwal, G. Juve, and E. Deelman, "Peer-to-peer data sharing for scientific workflows on amazon ec2," in *Proceedings of the 2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, ser. SCC '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 82–89.

[18] "Cloudfront," http://aws.amazon.com/cloudfront/.

[19] S. Pandey and R. Buyya, "Scheduling workflow applications based on multi-source parallel data retrieval in distributed computing networks," *Comput. J.*, vol. 55, no. 11, pp. 1288–1308, Nov. 2012.

[20] L. Ramakrishnan, C. Guok, K. Jackson, E. Kissel, D. M. Swany, and D. Agarwal, "On-demand overlay networks for large scientific data transfers," in *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, ser. CCGRID '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 359–367.

[21] D. Ghoshal, R. S. Canon, and L. Ramakrishnan, "I/o performance of virtualized cloud environments," in *Proceedings of the second international workshop on Data intensive computing in the clouds*, ser. DataCloud-SC '11. New York, NY, USA: ACM, 2011, pp. 71–80.

[22] S. Ostermann, R. Iosup, N. Yigitbasi, and T. Fahringer, "A performance analysis of ec2 cloud computing services for scientific computing," in *In ICST International Conference on Cloud Computing*, 2009.

[23] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan, "Interpreting the data: Parallel analysis with sawzall," *Sci. Program.*, vol. 13, no. 4, pp. 277–298, Oct. 2005.

[24] "Generic worker," http://www.venus-c.eu/Pages/Home.aspx.

[25] U. V. Çatalyürek, K. Kaya, and B. Uçar, "Integrated data placement and task assignment for scientific workflows in clouds," in *Proceedings of the fourth international workshop on Data-intensive distributed computing*, ser. DIDC '11. New York, NY, USA: ACM, 2011, pp. 45–54.

[26] B. Tiwana, M. Balakrishnan, M. K. Aguilera, H. Ballani, and Z. M. Mao, "Location, location, location!: modeling data proximity in the cloud," in *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, ser. Hotnets-IX. New York, NY, USA: ACM, 2010, pp. 15:1–15:6.

[27] "Azure caching," http://www.windowsazure.com/en-us/home/features/caching/.

[28] "Mef," http://msdn.microsoft.com/en-us/library/dd460648.aspx.

[29] "Ftp library," http://www.codeproject.com/Articles/380769.

[30] "Bittorrent library," http://www.mono-project.com/MonoTorrent.

[31] S. Sakr, A. Liu, D. Batista, and M. Alomari, "A survey of large scale data management approaches in cloud environments," *Communications Surveys Tutorials, IEEE*, vol. 13, no. 3, pp. 311–336, 2011.