A containerized analytics framework for data and compute-intensive pipeline applications

Yuriy Kaniovskyi
University of Vienna
Vienna, Austria
Yuriy.Kaniovskyi@
univie.ac.at

Martin Koehler
University of Manchester
Manchester, UK
Martin.Koehler@
manchester.ac.uk

Siegfried Benkner
University of Vienna
Vienna, Austria
Siegfried.Benkner@
univie.ac.at

ABSTRACT

The joint effort of scientific collaborations and the expanding data market creates demand for high-performance and data-intensive analytics infrastructures that can exploit the potential of heterogeneous multi-core architectures with dynamic and scalable execution environments. Contemporary approaches focus on developing efficient parallel application models, but lack the flexibility of efficiently integrating and utilizing native or accelerator-based code. In this work, we illustrate a novel approach on mending this shortcoming and offering seamless application integration into a highly versatile execution infrastructure. The centerpiece is a framework of containerized execution units and management thereof for satisfying the diverse requirements of data analytics pipelines and its stages. Containers not only ease distribution and deployment of applications, but, more importantly enable an efficient synthesis of different stage implementation variants aimed towards exploiting heterogeneous computing resources. Consequently, this approach allows the infrastructure to utilize mainstream data and compute-intensive techniques and paradigms to achieve the goal of efficient pipeline execution. We present our approach in form of a requirement analysis, a multi-tier architecture description, and deployment scenarios based on our current prototype implementation.

CCS Concepts

•**Computer systems organization** → **n-tier architectures;** •**Information systems** → *Data analytics;* •**Software and its engineering** → Application specific development environments;

Keywords

Data Analytics, Data Pipeline, Framework Design, Architecture, Container Virtualization, Implementation Variants, Optimization, Big Data, High Performance Computing

BeyondMR'17 May 19, 2017, Chicago, IL, USA

DOI: http://dx.doi.org/10.1145/3070607.3070613

1. INTRODUCTION

The efforts of contemporary scientific collaborations in various research fields lead to a high demand for large-scale computing and data-processing systems[17]. In recent years, scientific applications, ranging from particle physics simulations to high-throughput DNA sequence analysis, harnessed the power of high performance computing for simulation of complex formal models. These domains recently began leaning towards data-intensive computing, as the available data volumes outgrow computing capabilities [15].

The rapidly expanding scientific data market creates demand for data-intensive computing paradigms that meet the challenge of extracting knowledge from an ever-growing volume of data in a timely and useful manner, while supporting the analytic complexity and the requirements of the associated applications [3]. In view of the nature of scientific domains and their corresponding applications, there is a trend for developing high-performance data analytics infrastructures that can exploit the performance potential of heterogeneous multi-core architectures with flexible and scalable execution environments. Such environments provide scientific applications with sufficient computing power by combining conventional multi-core CPUs with various types of accelerators, such as GPUs or co-processors, offering a high degree of parallelism. Management and coordination of such environments is not a trivial task. The next generation of high-performance computing approaching exascale dimensions (with its own extremes and challenges), and with a growing demand for intensive data processing and analytics, multi-core execution environments are additionally required to cope with orchestrating data in relation to its distribution, storage and timely processing [12].

Many existing approaches and technologies tackle different issues and challenges with regard to high-performance data analytics. Stratosphere [2] enabled scalable data processing and provided an efficient execution engine supporting automatic parallelization and optimization aiming to maximize throughput and minimizing latency. Google's DataFlow [1] introduced a new programming model for efficient and scalable data pipeline applications, while offering on-demand resource provisioning and automatic orchestration of the pipeline execution tasks. Others, such as Intel, focused their efforts on integrating high performance and high throughput storage facilities [10] for Big Data applications, with the aim of minimizing bottlenecks and coordinating data movement and partitioning more efficiently. These frameworks focus on developing efficient parallel application models with trade-offs regarding, for example, replica-

tion and re-computation versus sharing of resources and the granularity of tasks. While they focused on efficiently taming unbounded data sets and deliver results in an interactive manner, we remark that such frameworks often require to rebuild applications from scratch based on the provided framework application model. While offering considerable efficiency, such an approach demands detailed knowledge of the model, the system and its underlying mechanics, making it difficult to integrate existing and accelerator-based code into the execution environment.

We offer a novel approach on improving flexibility, while supporting seamless application integration. The centerpiece is a framework of integrated and containerized execution units and management thereof for satisfying the diverse requirements of individual pipeline stages. In this context, lightweight virtualized Linux containers that already made an impact on the HPC community [11, 13] are essential to our approach. They not only accelerate development, ease distribution and deployment of applications, but enable an efficient synthesis of different pipeline stage implementation variants required for tapping heterogeneous computing resources. In addition, this containerization allows integration of different programming paradigms, such as MapReduce, OpenCL, MPI and CUDA, to achieve an optimized and resource-targeted application execution. The overall goal of the framework is to ease application deployment and management on heterogeneous architectures, and provide means for achieving high throughput by covering different requirements of individual data pipeline stages.

In this work, we illustrate the requirements, design and the architecture, based on our early prototype implementation of the data analytics framework for data and compute-intensive pipelines. The scope of challenges faced by such a system include the need to

- utilize generic execution mechanisms for building a flexible data analytics pipeline execution infrastructure;
- support the containerized integration of relevant application stages, to
  - enable use of a multitude of implementation variants and programming paradigms that satisfy diverging software needs of individual pipeline stages and
  - isolate pipeline stages to achieve balancing and sharing of computational resources across modern and emerging parallel and heterogeneous architectures;
- manage large-scale compute and data-distribution.

Contributions presented in this paper concern the design and architecture of the compute and data-intensive analytics framework aimed towards addressing the challenge of satisfying the diverging requirements of pipeline stages and supporting their deployment on distributed heterogeneous resources. The framework tackles this challenge by introducing generic mechanisms for native and accelerator-based code integration and support for different programming paradigms. These mechanisms encompass the containerization of pipeline stages and their runtime environments into isolated execution units. The goal is to realize a dynamic data analytics infrastructure that is able to efficiently orchestrate pipeline stage executions based on the stage's requirements and overall system utilization. Additionally, we briefly touch upon our approach for adaptive execution plan strategies and system descriptors required for the purpose of efficient execution unit orchestration.

The remainder of the paper is structured as follows: in the next section we describe the principal requirements of the proposed framework outlining the integrated pipeline application, the resource contention and application integration problems, and our approach on solving them. Section 3 depicts the layered high-level architecture of the framework describing components and integrated frameworks, necessary to satisfy the specified requirements. In Section 4 we illustrate concrete deployment and interaction scenarios based on the early prototype implementation that showcase how different components of the framework work in conjunction to achieve the set goals of the framework. Finally, we wrap up with concluding remarks, related work and future plans.

2. REQUIREMENTS

In the following, we discuss in detail the requirements of our data analytics framework with regard to its integrated application approach, execution environment and resource management. In general, the aim of the framework is to efficiently execute pipeline applications, taking into account the cost of large-scale data processing while providing the means to support flexible pipeline stage execution and utilization of heterogeneous resources.

2.1 Pipelined approach

In context of this work, we consider data pipeline applications as our primary use-case, as this type of applications is typically throughput-oriented and commonly used in the field of data analytics. A data pipeline follows the assembly line principle. It consists of a series of data processing modules, each possibly retrieving input data from multiple input pipes and providing results to several output pipes. Modules are interlinked by their corresponding input and output pipes. They thus form a workflow of data processing stages. Achieving high throughput requires key design considerations to focus on computational workload balancing of individual stages and achieving a fast delivery of data to and from them.

2.2 Diverging pipeline stages

A data pipeline may be composed of multiple, computationally highly divergent stages, each possibly utilizing specific programming frameworks and compute patterns. For example, it is common to have time-critical stream processing steps followed by data-cleansing and transformation, followed by highly-parallel, long running data analysis batch jobs. Each of these phases of a pipeline may employ its own programming framework and computational patterns with different resource demands. If multiple of such pipelines or stages are executed in parallel on a given set of resources, this poses a challenge for performance of individual tasks as well as overall system utilization.

2.3 Resource isolation

It is important to note that many scientific applications are designed for exclusive resource usage only. For example, it is common to deploy scientific applications on clus-

ters or supercomputers, where a batch-scheduler reserves the full range of the systems' resources for a specific timeframe. In case of data analytics pipeline applications (e.g. as is the case with the integrated data analytics pipeline described in Section 3.1), it is plausible to assume that a data pipeline may reserve all available computational resources and schedule them internally among its different stages. However, since exclusive resource usage may be highly inefficient, resource sharing is a common technique for enhancing performance and balancing system workload. We aim our framework implementation towards a more holistic approach of coordinated and efficient multi-framework and multi-task execution. Consequently, the framework requires individual data processing stages of a pipeline to be isolated as self-contained execution units in order to address resource contentions efficiently. This goal reflects the need for a flexible and high-performance compute infrastructure, that leads to interoperability, improved global system utilization, reduced time-to-deployment, reproduceability and portability of the given pipeline.

2.4 Implementation variants

The complexity of modern computing hardware entails a staggering diversity in programming environments, programming paradigms, libraries and supportive tools that may be applied to a data pipeline and its stages. In addition to an increase in programming complexity, application deployment as well as efficient execution on different, often highly heterogeneous systems, is a great burden for application developers. Hence, lightweight deployment models that encapsulate whole systems as self-contained executable units are essential to this work. Lightweight container virtualization can alleviate many of the challenges and requirements with regard to portability, scalability, deployment and optimization in resource utilization. The near bare-metal performance [6] compared to conventional virtualization techniques makes them suitable to be utilized in HPC-oriented environments. Linux containers not only provide the mentioned benefits to the framework and the integrated pipeline, but, more importantly, enable a seamless integration of native code (even in different programming languages) and use of different implementation variants of the pipeline stages optimized for different types of execution units (e.g., multi-core CPUs, GPUs, accelerators). The presented framework aims to adaptively optimize analytical pipeline jobs by evaluating efficient execution plans based on performance-relevant aspects of stage-specific implementation variants and their execution context, as well as available resources. The most efficient implementation variant may, however, imply a resource conflict that must be resolved by the framework in a way that maximizes the overall pipeline performance. To this end, we argue that it is beneficial to offer several variants for each pipeline stage. From a user's perspective, containerization of pipeline stages offers the means for specifying the pipeline sequence on an abstract level, which is then mapped by the framework to a concrete sequence of implementation variants of the pipeline stages. The use of containerized execution units enables the framework to decouple pipeline specification from its concrete implementation variant use, which offers a greater flexibility to the execution model.

In addition to supporting different implementation variants, we argue that some scenarios of the pipeline require utilization of specific programming paradigms to achieve the required efficiency. This is, for example, the case when there is a global data filtering or transformation needed prior to a complex computation of the available global data pool. Due to the considerable size of available data in such scenarios, the framework requires to support data-intensive processing techniques (e.g. MapReduce) for minimizing costly data transfers by co-locating computational units in the vicinity of the data (exploiting the data locality principle). Additionally, the goal of supporting heterogeneous architecture requires the framework to take benefit from the diverse accelerators present in current and emerging architectures. As such, HPC-bound programming paradigms such as OpenMP, OpenCL and CUDA are to be supported by the execution infrastructure.

2.5 Resource usage and optimization

Even for experts, the task of integrating, configuring, orchestrating and optimizing data and compute-intensive pipelines is a complex and time-consuming task that requires detailed knowledge of the underlying hardware and software resources. Consequently, current approaches are often static (i.e. optimizations are performed at design time), restricted to a single application or the pipeline layer only, and often assume a fixed execution environment. We therefore argue, that the framework needs support of adaptive execution mechanisms that are able to automatically assess the underlying computing and storage infrastructure, configure the execution environment according to available system resources and coordinate the execution of data pipeline stages based on their computational complexity and data-intensity.

2.6 Compute and storage resources

The framework is designed to be deployed on top of current and emerging distributed heterogeneous compute resources. This implies it to be aware of a multitude of system properties, including the amount of compute nodes and their computing resource capacity (CPUs, GPUs, and accelerators), as well as storage. As is the case with implementation variants, decoupling data from its specific storage location allows the integration of different storage platforms for the purposes of constructing a systematic memory hierarchy for the framework application in order to minimize costly data transfers and enable a strong fusion between the pipeline stages and its associated data sets. Large volumes of data necessitate the use of a distributed file system or a distributed data store, enabling support for data-intensive processing techniques and supporting large-scale data storage and data load balancing across the execution infrastructure. The local file system can be used to share local datasets across multiple, possibly related, local execution units of the job. Finally, in-memory storage can be used to either accelerate data passing between tasks running on the same computational node or hold data required throughout multiple pipeline stages. One of the objectives of the framework in this context is to achieve the highest possible data locality for a pipeline stage without neglecting its compute requirements. Consequently, the execution platform aims to place containerized pipeline execution units in the vicinity of the associated data sets.

The presented framework requirements allow scientists and developers to deliver pipeline stages in a modularized and self-contained manner. Through containerization, these

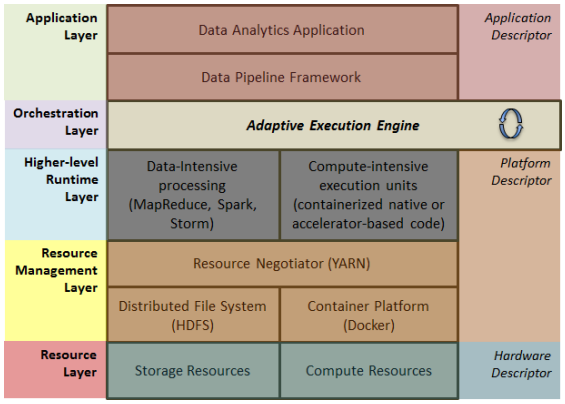

Figure 1: Framework architecture for data and compute-intensive data analytics, depicting different layers (left), their respective technologies, integrated frameworks or components (middle) and the associated system descriptors (right). The proposed framework comprises of an application layer, where the integrated data pipeline is staged; the orchestration layer, where the execution engine coordinates local and remote stage executions; the higher-level runtime layer, which integrates different execution platforms; the resource management layer, which integrates a resource manager to negotiate for available resources; and the (hardware) resource layer.

modules may include all the necessary tools and environmental configurations required for their execution. The end-user, shielded from concrete execution details and busy framework interactions, has to provide the pipeline specification and its data sets on an abstract level only in order to start an execution. Mapping an abstract execution plan to a concrete one is the responsibility of the framework. In order to fulfill these requirements, we illustrate the system architecture with its associated layers and interaction in the following section.

3. ARCHITECTURE

The high-level framework architecture for data and compute-intensive analytics incorporates different layers describing the data pipeline specification and implementation, the orchestration of pipelines, the supported runtime systems, resource management, and compute and storage resources available in the system. Figure 1 provides an overview of the architecture of the proposed framework. It illustrates different layers (left), their respective technologies, integrated frameworks or components (middle) and the associated system descriptors (right). In the following sections we discuss each layer, its responsibilities and core mechanics based on the specified requirements.

3.1 Application layer

Currently, the prototype integrates a generic and modular data pipeline used in the transportation domain [20] as its application layer and aims to enable it for scalable and responsive big data analytics. The pipeline provides a variety of modules for data integration (to and from different data sources), data transformation and data processing functions, which can all be arranged as a sequence of piped stages into a pipeline application. A modular plugin mechanism supports the development and integration of different implementation variants and additional application-specific modules. This mechanism was used to add Import-Buffer and Export modules enabling the pipeline to have access to the distributed file system.

The application layer consolidates application knowledge through the use of metadata descriptors. As part of the description model (depicted on the right hand-side in Figure 1), the application description encompasses information regarding available implementation variants (through the Variant Descriptors) of individual stages and data set mappings to the corresponding storage platform (through Dataset Descriptors). A pipeline execution request is specified in a YAML format[1], comprising a sequential chain of stage descriptions. Each stage description includes the name and the stage archetype tag (e.g. import, filter, transformation, function, custom), including a list of input and output pipelines. The execution ordering is set through specifying and linking input and output pipelines. Optionally, the user can also specify application or stage specific (e.g. custom computational weight) parameters and job priority.

The pipeline execution request is submitted using the framework client, which passes it onto the next layer for evaluation. The evaluation produces a concrete pipeline execution plan, which is used to initialize the deployment of the pipeline and its stages. An example of this process is illustrated in Figure 4 and discussed in Section 4.

3.2 Orchestration layer

The adaptive execution engine coordinates the execution of pipeline stages. This component of the framework bridges the gap between the higher-level runtime environments (and the associated task execution platforms) and the pipeline application. The management and coordination of the pipeline executions is facilitated through an execution plan compiled and later on adjusted by the requirements of the stages by the adaptive execution engine. An execution plan is characterized by: (1) a deployment and execution plan for pipeline stages and their implementation variants with respect to their computational requirements, and (2) a data management plan that evaluates the cost of data transfer based on the execution plan. An estimation of trade-offs between data transfer costs and performance and speedup penalties would optimize the pipeline execution, if the data set is large enough. While it may be debatable whether a global application analysis and decision-making – as opposed to a more dynamic stage-by-stage evaluation at runtime – may yield a better performance, we argue that the latter option provides a far more flexible execution model, when it comes to resource competition between stages, changing environmental (node-failure, accelerator availability) or stage-specific (data input/output amount) properties at runtime.

The framework implements a description model, that supports the evaluation of execution plans within the adaptive execution engine. The description model represents the main characteristics of system layers and components. These include a description of the underlying hardware, the execution platform and the respective pipeline variants, the characteristics of the data sets and sources and the application: the hardware descriptor comprises information about

[1]YAML Aint Markup Language (YAML) Version 1.2: http://yaml.org/spec/1.2/spec.html

the systems' compute and storage resources; the platform layer comprises the available execution and storage platform; and the application layer comprises descriptions discussed in the previous section. By taking this knowledge into account, and by using historical data for evaluation, the execution engine compiles an execution plan by using performance metrics that enable prediction of (relative) performance aspects of the execution units. These metrics can be provided by models producing a performance description according to environmental and execution specific characteristics, or can be specified by system experts. Performance modeling mechanisms can be implementation of a variety of different evaluation approaches, including analytic methods, methods that rely on historical performance data, heuristics, or any combination of these approaches [9]. Ranking of different execution solutions could be achieved through statistical estimation models. For example, in our previous work [14] we used the utility function to determine performance ranking of Hadoop job configurations based on a small set of system properties.

Finally, when a concrete execution plan (comprising a pipeline of stage implementation variants and their deployment assignments) is selected by the execution engine, it initiates deployment of the data pipeline and starts the resource negotiation procedure with the resource management layer.

3.3 Higher runtime layer

The higher runtime layer incorporates technologies and frameworks that satisfy the requirements of supporting multiple implementation variants and the required programming paradigms. On one end, the higher-runtime layer integrates frameworks that support established data-intensive processing techniques. One of the most important and well-known approaches in recent years has been the MapReduce [5] programming paradigm. MapReduce is typically utilized for analyzing Big Data sets stored on a distributed file system in a massively parallel and resilient manner. The extensively evolved ecosystem around the open source framework implementation Hadoop[2] and its decoupled resource negotiator YARN (Yet Another Resource Negotiator) [21], used within the framework, allows for other data-intensive analytics frameworks, most notably Apache Storm and Spark [23], to be integrated to provide additional data-intensive stage variants.

Additionally, the framework integrates a container virtualization platform for the purpose of bundling data pipeline stages (and possibly other applications or components) and their utilized runtime environment into self-contained, executable units. We chose to integrate the widely-used Docker[3] as our container platform. This choice is motivated by proven success in HPC environments, low overhead and easy integration with available resource management and computational frameworks (see next section). In addition, the simple mechanics for building Docker container images via the *dockerfile* not only allows developers of the data pipeline to provide additional templates with ease and on-the-fly, but its extensibility supports creation of modular templates, which further eases stage creation and delivery. For example, docker templates that are reserved for GPU execution have a template that includes a device driver forwarding from the host. With the internal docker registry made available to the execution engine and the resource management layer, docker templates are deployed as self-contained pipeline stage execution units.

[2]http://hadoop.apache.org/

[3]https://www.docker.com/

3.4 Resource management layer

The resource management layer incorporates technologies that ease the task of orchestrating complex pipeline executions for the adaptive execution engine. A resource negotiator enables partitioning and allocation of available resources required for a pipeline execution. The execution plan, compiled by the adaptive execution engine, is passed to the resource negotiator, which then starts the process of resource negotiation and allocation, deployment and the execution. After executing a specific pipeline stage, the resource manager delegates performance-relevant information back to the execution engine, in order for it to store this information for future evaluation of pipeline executions.

We assessed two resource negotiators for this layer. As mentioned above, YARN is a resource negotiator based of Apache Hadoop. YARN decouples the programming paradigm of MapReduce from its resource management capabilities, and delegates many scheduling functions (e.g., task fault-tolerance) to per-application components.

Mesos[8], similar to YARN, is a fine-grained resource negotiation engine that supports sharing and management of a large cluster of machines between different computing frameworks, including Hadoop, MPI, Spark, Kafka, etc.

The main difference between YARN and Mesos is the resource negotiation model. Whereas YARN implements a push-based resource negotiation approach, where container deployment requests specify their resource requirement and deployment preferences, Mesos uses a pull-based approach, where the negotiator offers resources to the container that it can accept or decline. The Mesos model is arguably more flexible, but requires more overhead due to its negotiation procedure. Furthermore, YARN's model for specification of resource and deployment preferences provides a greater control over resource partitioning across the system. Thus, we integrate Apache YARN to support the coordination and execution of containerized pipeline execution units, in addition to the aforementioned benefits of integrating different data-intensive frameworks.

The prototype implementation of the framework utilizes Apache YARN's resource and execution management mechanisms[4] to deploy and start containerized execution units on top of the infrastructure.

3.5 Resource layer

The resource layer comprises descriptors for compute and storage resources. Compute resource descriptors capture topological characteristics of the available hardware resources, including a description of compute nodes, their processing power, I/O and network devices. Additionally, the compute resource descriptor holds information about various PCI devices such as GPUs, Xeon Phi and other accelerators. Compute resource descriptors are specified using existing approaches - PDL [18] and hwloc [7]. In addition, we explicitly represent storage resource descriptors complementing compute resource descriptors with information about the

[4]https://hadoop.apache.org/docs/r2.7.2/hadoop-yarn/hadoop-yarn-site/DockerContainerExecutor.html

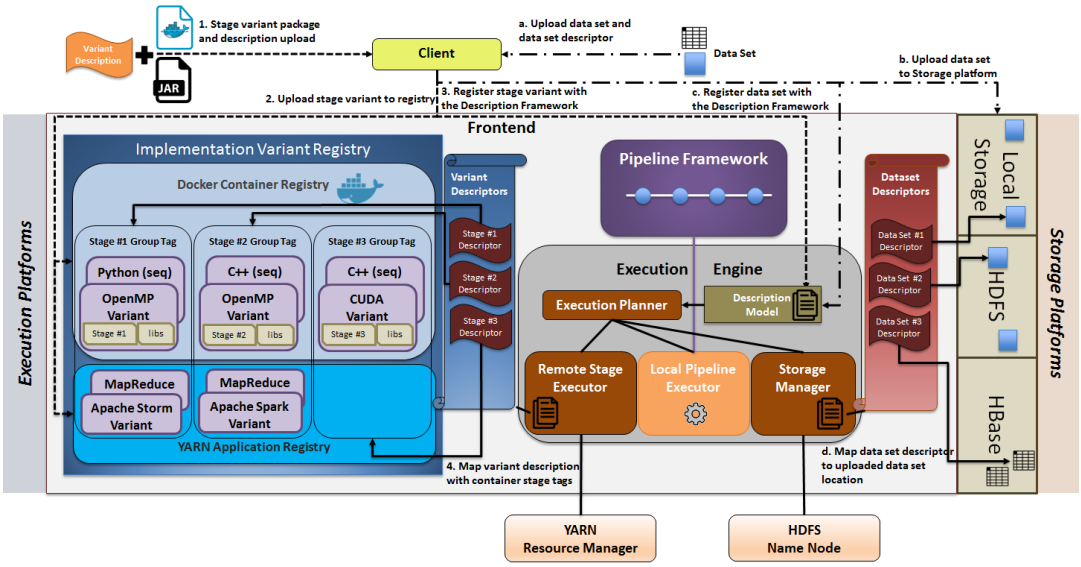

Figure 2: The frontend represents the control node of the compute and data-intensive pipeline execution framework. A client is used to upload new stage implementation variants and data sets, with their associated descriptors. Stage variants are added to the implementation variant registry, while data sets to the specified storage platform (HDFS per default). The frontend deploys the framework's execution engine, which coordinates pipeline execution and remote stage deployment.

memory hierarchy, in particular on cache, memory, disk and possible attached remote storage resources.

The storage resources may comprise a set of integrated storage platforms annotated by their associated descriptors to map access endpoints for the execution engine. Since we chose to utilize YARN as our resource manager, it was evident to utilize HDFS as our distributed file system. As such, we assume, that the bulk of the data is uploaded to the HDFS. To provide Docker containers with their corresponding data, the prototype framework implementation uses *MountableHDFS* and *libfuse*[5] to mount the HDFS data set structure (e.g. directory) to the local container file system. Docker instances are configured to map their data volumes[6] to the mounted HDFS data structure.

Through subsequent dependencies of the descriptors we are able to represent knowledge about the overall execution infrastructure and navigate from a specific job execution to its environmental characteristics.

4. DEPLOYMENT AND EXECUTION SCENARIOS

Deployment and execution scenarios outline the capabilities of the data analytics framework. Herein, we focus on showcasing how stage implementation variants are integrated, how a pipeline execution with different stage implementation variants is initiated, deployed and executed, and how data is managed across the execution infrastructure.

4.1 Pipeline stage and data set integration

The data and compute-intensive pipeline execution framework integrates the collections of pipeline stages and its implementation variants in form of a registry, as depicted in Figure 2. To add a stage variant, the user provides the stage variant package, which either consists of a *dockerfile* and its associated environmental setup or a YARN-native application in form of a *jar* (java archive) to the client. In addition, the Stage Descriptor, which annotates the stage variant for evaluation and utilization within the framework has to be provided. The descriptor consists of a stage group tag denoting the stage archetype (e.g. filter, transformation, computation, etc.), its execution platform (YARN or generic container-based), the variant paradigm or programming language (OpenMP, CUDA, C++), a custom-set computational weight (as opposed to the computational weight evaluated by the framework), and any application-specific parameters as key-value pairs. The implementation variant registry, representing the range of supported execution platforms, consists of two sub-registries: the first is the docker registry, which holds generic execution container images constructed from the *dockerfiles*. The images include the stage executable (in any native code) and the associated runtime, as well as required third-party libraries. The second sub-registry is the YARN application registry, which contains YARN-native applications. While this registry is primarily used for MapReduce-type stage implementations in our prototype, as mentioned earlier, other YARN-based stage implementations can be used here instead. Upon submission of a new stage variant, the client uploads the stage package to the corresponding registry and adds the new stage descriptor to the variant descriptors. Finally, the stage descriptor is mapped with an additional annotation to the location of the stage variant for the execution engine to quickly find it when required. The framework is then able to instantiate the newly adopted pipeline stage via the YARN's resource manager mechanics.

Data sets follow a similar procedure when added to the global data pool of the framework. A data set descriptor

[5]https://wiki.apache.org/hadoop/MountableHDFS
[6]https://docs.docker.com/engine/tutorials/dockervolumes/

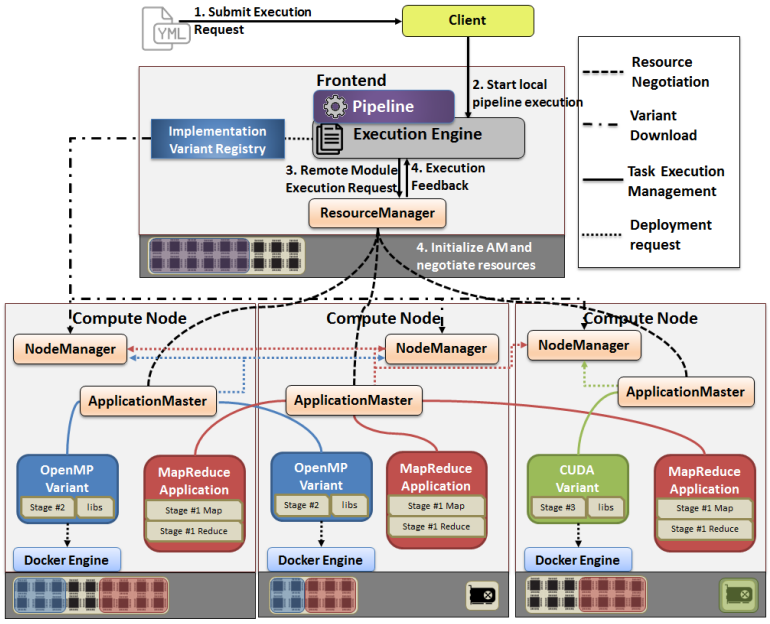

Figure 3: The scenario showcases the deployment of three remote pipeline stage executions on top of distributed heterogeneous resources. YARN's components aid in negotiating available resources for each individual pipeline stage, allowing their distribution across the compute nodes. The worker (NodeManager) nodes pull required images from the implementation variant registry to the computational node as requested by the ApplicationMaster - a controller for YARN tasks - and deploys them on the resources assigned by the execution engine.

is used to specify the storage platform, as well as related metadata such as size and location (e.g. path-to-dataset). If no storage platform is specified, the framework will utilize the HDFS as the global data pool, which is illustrated in the following scenarios. Using generic data set and storage platform descriptors supports the integration and utilization of a multitude of different distributed and non-distributed storage systems, such as HBase, RDBMS, in-memory DB, or the local file system. Upon submission, the client uploads the new data sets to specified storage platform and adds its descriptor to the description model. The process of integrating stages and data is illustrated in Figure 2 and denoted by interaction steps 1. to 4. and a. to b. accordingly.

The framework integrates a generic and modular data pipeline that allows the execution engine to deploy pipeline application instances. The execution engine uses a local pipeline executor to initialize and start the pipeline application locally, and a remote stage executor to deploy stages on remote heterogeneous nodes. The Execution Planner tries to optimize pipeline execution plans and coordinate the deployment of the pipeline and its stages with the support of the description model.

4.2 Deployment on remote computational resources

A pipeline application execution can be initiated by submitting an abstract specification of the pipeline via the client. Currently, this specification is a YAML file consisting of a sequence of stage archetypes connected via input-output pipes, the input data specification and optional application-specific parameters. Upon the pipeline specification submission, the execution engine evaluates the pipeline application in terms of its resource requirements as described in Section 3.2.

In the scenario depicted in Figure 3, the execution engine concluded the deployment of three remote stages: the first stage is deployed as a MapReduce application, the second is deployed as an OpenMP variant (e.g. filtered data set transformation) and the final one as CUDA (e.g. compute-intensive function). This concrete pipeline execution specification induces the *Remote Stage Executor*, a component of the execution engine, to issue YARN application deployment requests to the *ResourceManager*. Each stage of the pipeline is treated by YARN as separate application and is assigned to its own *ApplicationMaster* to negotiate for available resources. This allows for a greater resource isolation (no interference with whatever is installed on the host) of each stage, and a refined allocation based on priorities and resource requirements. The priority for resource negotiation can be influenced by the execution engine's evaluation or user specification (in the form of a custom computation weight). The *ApplicationManager*, a component of the *ResourceManager*, starts YARN's *ApplicationMaster* on a compute node. YARN's *ApplicationMaster*, as is the case with the default YARN setup, is responsible for deploying, executing and monitoring distributed stage tasks. It will establish a connection to the *ResourceManager* to negotiate and receive a set of resources for container deployment. In addition, the *ApplicationMaster* establishes a communication channel with YARN's *NodeManager*, which is essentially a worker for the *ResourceManager* (master). The *ApplicationMaster* directs the *NodeManager* to download the application container from the corresponding registry and deploy

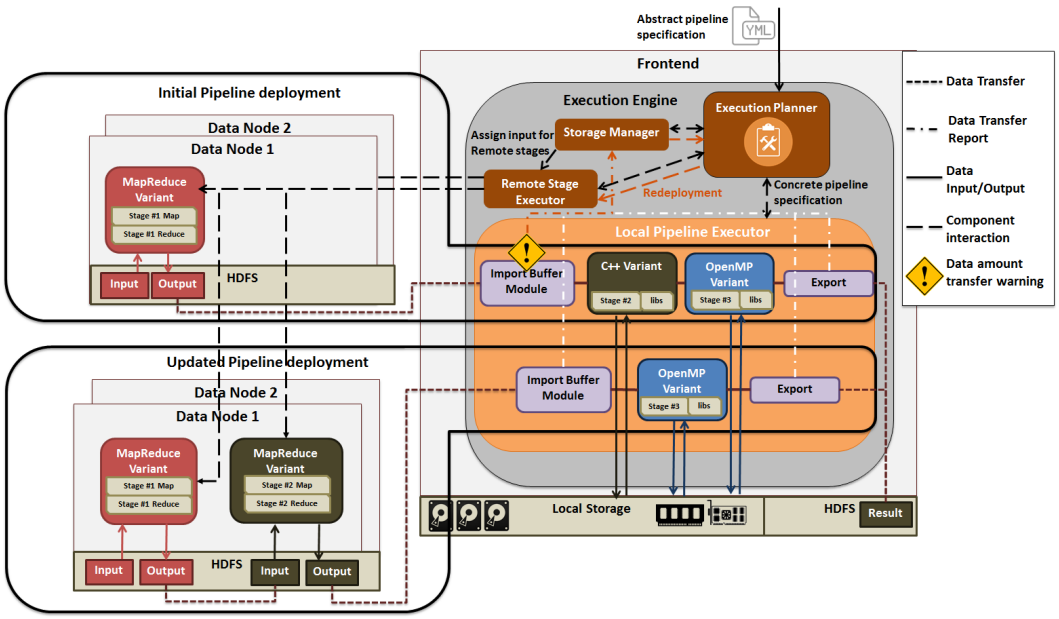

Figure 4: This scenario showcases how the execution engine deploys a concrete pipeline and handles issues related to data management. The initial pipeline execution plan dictates deployment of the first stage as MapReduce and all consecutive stages in the local pipeline instance. Due to an unexpected amount of data to be transfered to the local pipeline, the execution engine (prompted by feedback reports) initiates a reevaluation of stage deployment at run-time and updates the deployment plan, with the effect that stage two (unable to handle the amount of data efficiently in its initial state) is also deployed as a MapReduce.

it. In case of the first stage of this scenario, the MapReduce Application is deployed on all nodes, since data blocks for the computation can be co-located for processing on each HDFS node. The other two stages are deployed on a limited amount of compute nodes, as their computation weight supersedes their data-intensity. In this case nodes are chosen according to their resource availability. Finally, running stages are constantly monitored by the *ApplicationMaster*, and together with the *NodeManager*'s Resource Reports, the *ResourceManager* receives constant updates on the state of the stage execution in terms of resource allocation and task status. This information is used as stage execution feedback and stored to a historical execution database within the adaptive execution engine for evaluation of future stage executions.

It is important to note that in the current release of YARN, *NodeManager* uses a statically defined configuration file[7] that specifies the container type that the *NodeManager* has to deploy. The framework, however, requires these properties to be set dynamically - upon stage deployment. To work around this issue, we currently use a script to interrupt the *NameNode*, change its configuration and restart it in order to differentiate between Docker and YARN container deployment. In future we aim to extend the *NodeManager* in its capabilities of dynamically instantiating the requested container type.

[7]Natively YARN NodeManager configurations are statically defined in *yarn-site.xml* configuration file, with the following relevant properties: *yarn.nodemanager.container-executor.class* and *yarn.nodemanager.docker-container-executor.exec-name*

4.3 Dynamic data management

Figure 4 illustrates how the execution engine deploys a concrete pipeline and handles data management issues. In this scenario, the initial pipeline deployment plan foresees the deployment of the first stage (e.g. filter) through a MapReduce variant. Consecutive stages are to be deployed on the local pipeline instance. The execution planner thus provides the concrete pipeline execution plan to the local pipeline executor, which instantiates the pipeline application on the frontend. At the same time, the deployment specification of the first stage is passed to the remote stage executor, which submits application requests to YARN's *ResourceManager*. The Storage Manager is used to link the input data set of the pipeline and assigns this data set to the MapReduce job configuration. The special stages deployed within the pipeline application Import-Buffer and Export handle data transfer to and from the locally deployed pipeline. Upon data transfer, these stages submit reports on the amount of data to be transferred to the data manager.

In the scenario outlined in Figure 4, the processing of the MapReduce stage yields a considerable amount of data bound to be transfered to the next stage of the pipeline (C++-variant). The Import-Buffer communicates a warning, denoted in Figure 4 as a warning sign regarding the amount of data to be transferred. The Storage Manager, having evaluated the time needed for the transfer communicates this to the execution planner, as this data transfer may turn out to be excessive. The Execution Planner, being aware of the pipeline execution, concludes that the currently devised plan may impair the efficiency by this data transfer and performance of the consecutive stage. Following this reevaluation, the Execution Planner concludes that,

due to the availability of the second stage as MapReduce, that stage should be deployed as such. This causes the redeployment of the updated execution plan at runtime. The local pipeline executor is directed to re-instantiate the local pipeline execution according to the updated deployment plan, while the remote stage executor is directed to deploy the MapReduce variant of the second stage. The redeployment and execution of the updated execution plan generates no further warnings, as the amount of data can be handled without further issues.

While our framework focuses on reducing costly data transfer bottlenecks, we argue that the same technique may be applied to reevaluate compute-intensive stages, e.g. when accelerator resources on specific nodes become available. Since the Execution Engine is aware of the Pipeline application execution plan and the availability of resources, a reevaluation based on the computational grade of the consecutive pipeline stages may result in outsourcing them to more powerful compute resources. Such an evaluation has to consider the specific trade-off between data transfer time and computation time, while keeping the stage overall evaluation overhead to a minimum (using the aforementioned statistical evaluation methods).

In Section 3.2 we argued that evaluating the application stage-by-stage provides a more flexible execution model. However, we emphasize that some decisions require a broader scope of the execution plan. The framework prioritizes effective execution of compute and data-intensive stages, and deploys them on the appropriate resources if possible. Non-intensive stages may be reevaluated and redeployed based on the current state of the pipeline execution, such as data transfer in this scenario. Thus, a hybrid approach of evaluating the most pressing stages of the pipeline and run-time adaptation (depending on changing environmental properties) of less intensive stages may be appropriate. Such an approach may introduce additional and possibly excessive overheads. However, we argue that following our evaluation approach for pipeline execution plans, such redeployments may become less frequent as the framework adapts to its execution environment and newly adopted stage variants.

5. RELATED WORK

We have mentioned related approaches (Stratosphere, Dataflow) in the field of data analytics as our motivating example in Section 1. Here, we discuss a few of the other relevant approaches that tackle the challenges related to compute-intensive big data analytics.

KeystoneML [19] introduces an approach for large-scale pipeline optimization. The authors focus on capturing end-to-end pipeline application characteristics that are used to automatically optimize execution at both the operator and pipeline application levels. They enable their system to produce pipeline solutions that automatically adapt to changes in data, hardware, and other environmental characteristics.

Marcher [24] - a heterogeneous system for high performance computing and big data analytics applications supports for a wide range of mainstream parallel programming models (including OpenMP, OpenCL, CUDA, MPI and Map-Reduce) and diverse system resources on the basis of an extensive API and numerous programming interfaces. They focus their infrastructure towards energy-efficient low-level task execution.

Apache Spark [23], mentioned in this work as one of the possible integrated frameworks introduces resilient distributed data sets on top of Apache Hadoop. The framework supports iterative tasks and improves performance by explicitly specifying caching of distributed data sets. A wide range of functions support categorization of application components into data transformations and actions. In addition, Spark provides stream processing functionality and a machine learning library mllib.

Another popular effort towards efficient compute and data-intensive task execution is the MapReduce-MPI [16] library. The authors implement the MapReduce paradigm using MPI in order to achieve a scalable, data-intensive execution for performance-oriented graph algorithms.

The European PEPPHER project [4] proposed a component-based development approach for heterogeneous parallel systems allowing to relieve application developers of low-level implementation details, while providing means for a seamless integration of different programming APIs, as well as for dynamic code adaptation and optimization. Parallel applications are composed at a high-level of abstraction, while providing implementation variants, optimized for different processing units. While some concepts of PEPPHER are applicable in this work, such as utilization of different implementation variants, the requirement of rewriting all application modules to adhere to a high-level API is not feasible. Another approach in a similar direction is Elastic computing [22] for cloud computing. The work focuses on the provisioning of function variants, called elastic functions, among which the best combination is composed mostly by static means, guided by performance profiles and models.

In contrast to the aforementioned technologies, we focus on integrating containerized implementation variants to support mainstream parallel programming models and paradigms to exploit heterogeneous architectures. Our framework is designed to integrate different storage platforms and utilizes concepts from autonomic computing to realize efficient data analytics pipeline executions. In a broad sense, we combine some of the mentioned approaches and technologies and utilize their mechanics relevant for the requirements of our envisioned system.

6. CONCLUSION AND FUTURE WORK

In this paper, we presented requirements, architecture and usage scenarios of the compute and data-intensive pipeline execution framework. The framework design supports integration of a multitude of stage variants for the purpose of optimizing pipeline applications with respect to data and computational requirements of individual stages. The design requirements include support for diverging pipeline stages, which make it necessary to encapsulate them and their runtime environment into containerized execution units. This allows for a refined resource isolation and consequently an efficient resource utilization. More importantly, containers provide the means of supporting concurrent utilization of different implementation variants based on a multitude of programming languages and paradigms. The framework integrates a resource negotiator that aids in managing and orchestrating a generic set of execution platforms. We illustrated three scenarios related to pipeline stage and data set integration, deployment of remote pipeline stages and data management. As such, our framework supports a generic and flexible execution infrastructure based on container

technologies, able to exploit massively parallel data processing techniques on one extreme, and HPC-bound tasks on the other. Finally, the framework's adaptive execution engine, supported by the description model, which aggregates information on the current system state, aims to evaluate and coordinate execution with the main goal of achieving efficient processing of data and compute-intensive pipelines.

The current prototype supports the execution of modular data pipelines using different implementation variants. It integrates Apache YARN as its resource negotiator, Docker as its container platform and HDFS as its global data pool. The early implementation of the execution engine selects variants based on estimated (with regard to historic execution data) output data measurements. The framework prototype has several construction areas, some of which where mentioned in this work. Consequently, our main goal for the future is to fully implement the execution engine and provide a proper evaluation of pipeline stage executions on heterogeneous many-core architectures.

Acknowledgment

This research has been supported by the Austrian Research Promotion Agency (ICT of the Future) under grant agreement #845606 (Retida Project).

7. REFERENCES

[1] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, and S. Whittle. The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proc. VLDB Endow.*, 8(12):1792–1803, Aug. 2015.

[2] A. Alexandrov, R. Bergmann, S. Ewen, J.-C. Freytag, F. Hueske, A. Heise, O. Kao, M. Leich, U. Leser, V. Markl, F. Naumann, M. Peters, A. Rheinländer, M. J. Sax, S. Schelter, M. Höger, K. Tzoumas, and D. Warneke. The stratosphere platform for big data analytics. *VLDB*, 23(6):939–964, Dec. 2014.

[3] R. Arora. *Conquering Big Data with High Performance Computing*. Springer, 1st edition, 2016.

[4] S. Benkner, S. Pllana, J. Traff, P. Tsigas, U. Dolinsky, C. Augonnet, B. Bachmayer, C. Kessler, D. Moloney, and V. Osipov. Peppher: Efficient and productive usage of hybrid computing systems. *Micro, IEEE*, 31(5):28–41, Sept 2011.

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

[6] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio. An updated performance comparison of virtual machines and linux containers. *IEEE ISPASS*, 00:171–172, 2015.

[7] B. Goglin. Managing the topology of heterogeneous cluster nodes with hardware locality (hwloc). In *HPCS'14*, pages 74–81, July 2014.

[8] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 295–308, Berkeley, CA, USA, 2011. USENIX Association.

[9] M. C. Huebscher and J. A. McCann. A survey of autonomic computing - degrees, models, and applications. *ACM Comput. Surv.*, 40:7:1–7:28, August 2008.

[10] Intel, High Performance Data Division. Whitepaper: Big data meets high performance computing, 2014.

[11] D. M. Jacobsen and R. S. Canon. Contain this, unleashing docker for hpc, 2015.

[12] S. Jha, J. Qiu, A. Luckow, P. K. Mantha, and G. C. Fox. A Tale of Two Data-Intensive Paradigms: Applications, Abstractions, and Architectures. In *BigData Congress*, pages 645–652. IEEE, 2014.

[13] S. Julian, M. Shuey, and S. Cook. Containers in research: Initial experiences with lightweight infrastructure. In *XSEDE'16*, pages 25:1–25:6, New York, NY, USA, 2016. ACM.

[14] M. Koehler, Y. Kaniovskyi, and S. Benkner. An adaptive framework for the execution of data-intensive mapreduce applications in the cloud. In *DataCloud 2011*, Anchorage, Alaska, May 2011. IEEE.

[15] R. T. Kouzes, G. A. Anderson, S. T. Elbert, I. Gorton, and D. K. Gracio. The changing paradigm of data-intensive computing. *Computer*, 42(1):26–34, Jan 2009.

[16] S. J. Plimpton and K. D. Devine. Mapreduce in mpi for large-scale graph algorithms. *Parallel Comput.*, 37(9):610–632, Sept. 2011.

[17] D. A. Reed and J. Dongarra. Exascale computing and big data. *Commun. ACM*, 58(7):56–68, June 2015.

[18] M. Sandrieser, S. Benkner, and S. Pllana. Using explicit platform descriptions to support programming of heterogeneous many-core systems. *Parallel Computing*, 38(1-2):52–65, 2012.

[19] E. R. Sparks, S. Venkataraman, T. Kaftan, M. J. Franklin, and B. Recht. Keystoneml: Optimizing pipelines for large-scale advanced analytics. *CoRR*, abs/1610.09451, 2016.

[20] J. L. Toole, M. Ulm, M. C. González, and D. Bauer. Inferring land use from mobile phone activity. In *ACM SIGKDD'12*, pages 1–8. ACM, 2012.

[21] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler. Apache hadoop yarn: Yet another resource negotiator. In *SOCC '13*, pages 5:1–5:16, NY, USA, 2013. ACM.

[22] J. R. Wernsing and G. Stitt. Elastic computing: A framework for transparent, portable, and adaptive multi-core heterogeneous computing. *SIGPLAN Not.*, 45(4):115–124, Apr. 2010.

[23] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.

[24] Z. Zong, R. Ge, and Q. Gu. Marcher: A heterogeneous system supporting energy-aware high performance computing and big data analytics. *Big Data Research*, 2017.