Percipient StorAGe for Exascale Data Centric Computing

FETHPC1 - 671500

WP 4 Programming Models, Runtime and Data Analytics for Applications

D4.3 Report on Percipient Storage Run-Time System Architecture

D4.3 Report_on_Percipient_Storage_Run-Time_System_Architecture_1.0

Scheduled Delivery: 31.07.2016
Actual Delivery: 30.07.2016
Version 1.0

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 671500		
Dissemination Level		
PU	Public	X
PP	Restricted to other programme participants (including the Commission)	
RE	Restricted to a group specified by the consortium (including the Commission)	
CO	Confidential, only for members of the consortium (including the Commission)	

Responsible Partner: Forschungszentrum Jülich (JUELICH)

Revision history:

Date	Editor	Status	Version	Changes
10.7.2016	Dirk Pleiter	Draft	0.1	Initial Draft
13.7.2016	Dirk Pleiter	Draft	0.2	Complete draft for technical review
23.7.2016	Dirk Pleiter	Final draft	0.3	Update based on feedback from G. Congui, S. Narasimhamurthy and S. de Witt

Authors
Dirk Pleiter (JUELICH), Nicolas Vandenbergen (JUELICH)

Internal Reviewers
Giuseppe Congiu (SEA)

Acknowledgements
The research leading to these results has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 671500

More information
The most recent version of this document and all other public deliverables of SAGE can be found at http://www.sagestorage.eu

Glossary of Acronyms

Acronym	Definition
BGAS	Blue Gene Active Storage
EC	European Commission
HPC	High Performance Computing
LDAP	Lightweight Directory Access Protocol
MERO	Seagate's Object Storage software technology
NEST	Neural Simulation Tool
POSIX	Portable Operating System Interface
SAGE	Percipient StorAGe for Exascale Data Centric Computing
SKA	Square Kilometre Array

Table of Contents

List of Figures

List of Tables

1. Executive Summary

A principal goal of the SAGE project is, not only to develop usable hierarchical storage architectures based on the object store Mero, but also to enable exploitation of processing capabilities at different levels of the storage architecture. One of the primary purposes of the SAGE run-time system is to provide a service infrastructure that implements the generic functions required for task offloading, as well as application-specific data management tasks. The requirements are derived from a set of generalised use cases linked to different workloads within the SAGE application portfolio. Based on these requirements, the architecture is outlined and an implementation, as well as a test and validation plan, is provided. Based on the design of the SAGE prototype, a testing and evaluation plan is established.

Partner	Contribution in Person Months
JUELICH	3

Table 1 Partner Contributions for Deliverable

2. Introduction

This section describes the deliverable's objectives and approach, and defines the specific tasks undertaken, as well as the relation to other project tasks and WPs.

2.1. Objectives

The objectives of this deliverable are as follows:

- Document the use cases that drive development efforts within task T4.2 “Percipient Storage Run-time System”.
- Provide the list of requirements based on the analysis of the use cases.
- Outline the architecture of the run-time system that supports the identified use cases and meets the defined requirements.
- Define the plan for implementing the outlined architecture.
- Define the test and evaluation plan for the implementation that exploits the SAGE prototype.

2.2. Tasks Associated with the Deliverable

The following tasks are associated with the deliverable.

Task Number	Description of Task (as per plan)	Planned activity within task for this Deliverable (Brief Summary)	Performed activity and any deviation from Plan (Brief Summary)
T4.2	This task implements a run-time system allowing direct exploitation of the percipient features of the SAGE architecture for off-loading tasks	Formulation of a design specification based on experience with Blue Gene Active Storage (BGAS)	Unchanged from planned activity

Table 2 Tasks Associated with the Deliverable

2.3. Relation to Other Tasks and WPs

The work for T4.2, documented in this deliverable, is related to the characterisation of workloads performed within T1.1, whose results have been documented in D1.1. The results will, among other purposes, be used for T4.3 (“Visualisation”).

3. Use Cases

In this section we describe a set of use cases, that has driven the architecture outline presented in Section 5.

While each use case is linked to a specific workload, they can be considered as representative of different classes of applications. For instance, the use cases differ in terms of the source of the data that is being processed. In the following sections, we assume an HPC infrastructure with a scalable compute system attached to an I/O subsystem based on the SAGE architecture. Taking a compute system centric view, we can distinguish the following data flow cases:

- Data retrieved from an external source and read by the compute system for processing.
- Data created by the compute system and written to the I/O subsystem.
- Data written and/or read by the compute system, which uses the storage to extend the available memory condition.

In the description of the use cases, we will also use a methodology to characterize workflows, which we call retention time analysis [SAGE2016]. Within this analysis, the data products consumed inside the workflows are classified according to the following scheme:

- Transient (Temporary): Data discarded on simulation completion or when later processing steps are concluded.
- Short-term (Campaign): Data used throughout the execution of the scientific workflow.
- Permanent (Forever): Data outliving the machine used to generate it.

3.1. Semi-Persistent Data Cache

The first use case concerns workflows in which data products, which are created outside the HPC infrastructure, need to be read by an HPC compute system for processing. The rate at which these data products are generated is assumed to be sufficiently small such that they can be archived before being processed. Data processing is assumed to consist of managing a list of data products, in which each data product can be processed independently. These kinds of workflows occur in the context of observatories, such as satellites or radio telescopes.[1] Finally, we assume the time required to process one data product to be short as

[1] Although radio telescopes produce or, e.g. in the case of SKA, will produce data at a rate at which significant parts of the data cannot be stored, these telescopes are producing or will produce data products that require further processing.

compared to the time required to load the data product from the archive to the compute system. Additionally, pre-processing steps might be required before data is ready for processing.

The pilot application matching this use case is the Jülich Rapid Spectral Simulation Code (JURASSIC). It is a fast, radiative transfer model for the mid-infrared spectral region developed at the Jülich Supercomputing Centre (JSC) [JURASSIC]. JURASSIC is being used to retrieve information about the atmosphere from Atmospheric InfraRed Sounder (AIRS) measurements. AIRS is one of the instruments aboard the Aqua satellite [Aqua]. Its mission started in September 2002 and is expected to continue at least until the year 2020, resulting in large quantities of satellite data to be analysed by JURASSIC.

The retention time analysis for JURASSIC is shown in Figure 1. Initially, the data products are stored permanently inside an archive. To enable pre-processing and fast reading of the data into the compute system, the data products should be staged, i.e. copies classified as short-term need to be created, ideally before the job starts.

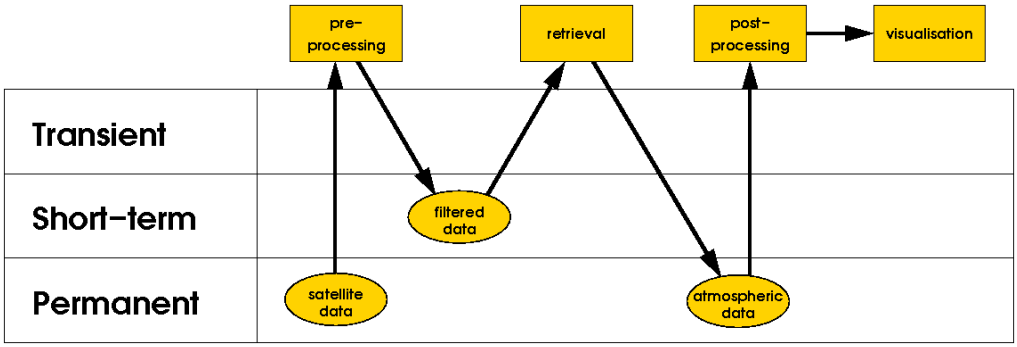

Figure 1: Retention Time Analysis for a Typical Workflow Using JURASSIC

In traditional workflows, a key challenge to staging data products is the lack of information about when processing will occur. Here, the storage system is only informed that a specific data product is required after the corresponding file is opened. As a read typically starts soon after the file is opened and as the time required for data processing is assumed to be short compared to the time required for data loading, read-ahead strategies are not expected to be effective. Several strategies are available to exploit hierarchical storage architectures for this kind of workflow. One option is to inform the HPC infrastructure's resource manager about the input files required to execute a given job. It is up to the resource manager to orchestrate the staging of data into a fast storage tier and to schedule the job once data is available. This approach is, e.g. currently implemented at NERSC [NERSC2016]. Within SAGE, we propose to take a different approach, which we call semi-persistent data cache.

The semi-persistent data cache employs a data-centric approach in which an in-storage service manages data staging. Instead of processes on the compute system deciding which data product to process next, our approach foresees a mechanism for these processes to retrieve information about data products that are available and ready for processing.

3.2. Data Post-Processing

The next use case concerns data post-processing, in which data is created by an application running on a highly parallel compute system. The application is assumed to produce data at such a high rate that the bandwidth to a lower, capacity-optimized storage tier would become a bottleneck. Post-processing of data at a higher storage tier would allow for a significant reduction in data volume, as only the post-processing results need to be stored, thereby mitigating the aforementioned problem. Alternatively, in-situ data analysis concepts could be used. These would, however, make it difficult to perform post-processing asynchronously or even interactively. Furthermore, in-situ data analysis might suffer from scalability issues, depending on the type of analysis that needs to be performed, while in-storage data post-processing can be assumed to be performed on a significantly smaller number of processing devices; the degree of parallelism is significantly reduced.

A driving application behind this more generalized use case is the analysis of data produced by spiking neuronal network simulations. The Neural Simulation Tool (NEST) is such a simulator [NEST, NEST2007]. It focuses on the dynamics, size and structure of neural systems rather than on the exact morphology of individual neurons, which allows for scaling to very large network.[2] An extended retention time analysis is provided in Figure 2, showing a simulation that generates output data requiring post-processing as well as output data that is stored permanently. The simulation output for post-processing is transient, as it can be destroyed once post-processing is completed. The data products generated during post-processing are stored in a permanent storage tier.

[2] The size of the network is limited mainly by available memory capacity.

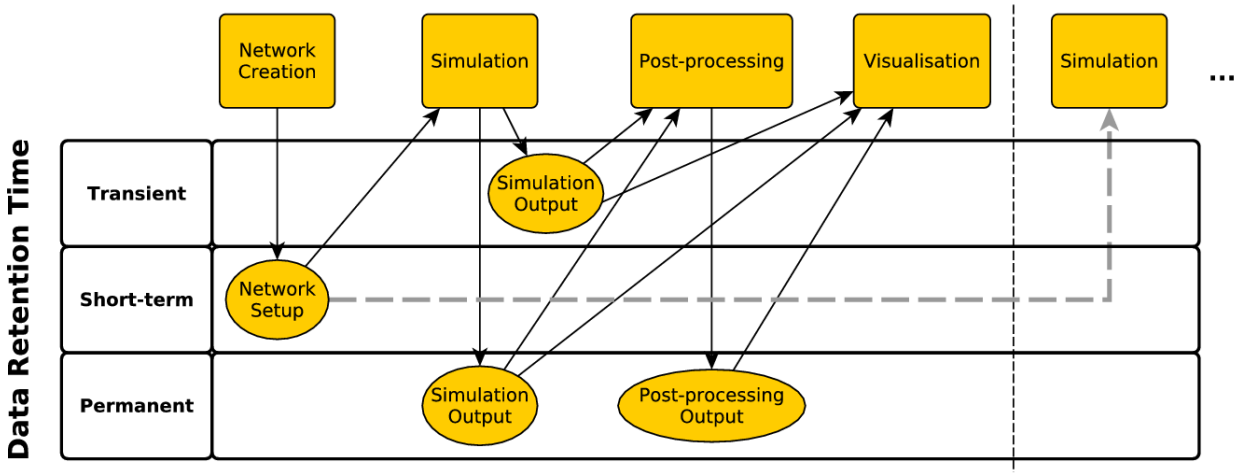

Figure 2: Retention Time Analysis for a Workflow Using the NEST Simulator

NEST is performing task-local I/O and thus, each rank of the application creates one or more objects while the simulation is executed. During post-processing, applications written in C or C++ or scripts written in Python are executed to process data. In order to aggregate results, data-suitable communication mechanisms are required.

3.3. Virtual Memory Hierarchy

The final use case considered within this task is not linked to a specific application, but rather to a more generalized concept of managing data for visualisation. The data volume is assumed to be extremely large as compared to the capacity of the memory directly attached to the rendering devices. The capacity of the device memory in today's high-end graphics cards is in the order of 10 GByte, while the size of data sets to be visualised may reach the order of 10 TByte. The high performance of graphics cards has to be balanced by extremely high bandwidth towards device memory. The challenge of providing high-bandwidth access to data, as well as high capacity to hold large data sets, can be tackled using a virtual memory (or storage) hierarchy in which data movement is managed by software. Using a software-managed cache approach was first proposed by Fogal et al. [Fogal2010] and later used for petascale microscopy data streams by Hadwiger et al. [Hadwiger2012]. This approach has been extended in the project "Interactive Volume Rendering of Massive Data on the Blue Gene Active Storage Architecture" of JUELICH and RWTH Aachen.

4. Architectural Requirements

In this section we collect a set of architectural requirements for implementing the use cases described in the previous section.

4.1. General Requirements

1. Resource discovery: Ability to discover the resources available for a given realisation of the SAGE architecture.
2. Resource management: Ability to interact with a resource management system, handling available storage and compute resources.
3. Process management: Availability of a service infrastructure for starting, controlling and ending processes within the SAGE architecture.
4. Centralised logging infrastructure: Capability of communicating logging messages created by different components of the run-time to a central location.
5. Debugging capabilities: Capability of the system to enable debugging of the behaviour of applications running within SAGE.
6. Protection against the system becoming compromised, including acquisition of the wrong privileges.
7. Quality of service management: Mechanisms to prevent SAGE system functionalities becoming blocked or significantly slowed-down by actions triggered through the runtime system.
8. Communication channels: Compute node process must be able to communicate with processes running within SAGE and the latter must be able to communicate among themselves.
9. Secure communication mechanisms: Mechanisms that allow for communication between compute system processes in user space and services running within the SAGE architecture ought to be sufficiently secure to protect the system from becoming compromised.
10. User management integration: Access to user mapping information.

4.2. Use Case Specific Requirements

1. Availability of a semi-persistent cache manager into which a workflow system can connect.
2. Availability of a framework that allows:
    - Deployment of user code within SAGE
    - Spawns processes for object post-processing and monitors their execution
3. Availability of a virtual memory management system capable of:
    - Loading data from lower levels of the virtual memory (or storage) hierarchy
    - Providing available data to higher levels

- Managing data placement and virtual cache evictions

5. Architecture Outline

In this section we outline the architecture of the SAGE runtime system, and describe the main components and their functionality. We will start by describing a set of services and functions that provide the basis for a distributed, event-driven service infrastructure. After considerations related to security and resource management, we outline the use case specific services and functions.

5.1. Base Services and Functions

The services and functions described in the following subsections are required to establish a base infrastructure that are required by any of the more specialised services described in subsection 5.3 to 5.5.

5.1.1. Management Service

The management service is implemented by daemons running on every SAGE node. To enhance robustness, its functionality is kept to a minimum. One of the management daemons will be configured to act as master. It is the only service that needs to be statically configured such that any other management service can dynamically register with the master management service. The management services run with a privileged user ID.

After start-up, a management daemon will go through an initialisation phase comprised of the following steps:

- Discover locally available resources
- Unless it is the master daemon, register with the master daemon and communicate available resources

After initialisation, the management daemons are ready to:

- Receive messages
- Spawn runtime service processes
- Monitor spawned processes
- Implement the logging infrastructure

5.1.2. Process Spawning

Other runtime system processes are started by the local management daemon using a process fork followed by the creation of a new process image using one of the POSIX exec functions [Linux-exec]. The management daemon continues to monitor the status of spawned processes and collect the exit status.

5.1.3. Communication Framework

We assume that processes running on the SAGE nodes and on the attached compute nodes can communicate via TCP/IP (i.e., IPoverIB) and thus, a communication infrastructure for sending messages that comprise event notifications, with or without data, can be implemented using a standard message queuing technology, such as ZeroMQ [ZEROMQ].[3] This message queuing technology will also be used for intra-node communication between different service processes as well as between different threads of a service.

5.1.4. Logging Framework

To facilitate logging of critical events, a hierarchical logging scheme will be implemented as follows:

- All processes started by a management daemon send log messages to this management daemon.
- The management daemons send or forward log messages to the master management daemon.
- The master management daemon saves all log messages.

5.1.5. Resource Discovery

During resource discovery, the following information is collected:

- SAGE storage tier level[4]
- Capacity of the locally available storage resources
- Compute capabilities

5.2. Security and Resource Management

One challenge in building a distributed service infrastructure to which different users can connect, is to control the identity of callers.[5] We start with the assumption that SAGE nodes and compute nodes are integrated into the same user management system, which could, e.g. be based on LDAP. This allows exploitation of the SASL PLAIN mechanism available in ZeroMQ for initial authentication [SASLRFC, ZEROMQ]. As this approach has limitations, alternatives will be explored within this project. This implementation, however, is probably

[3] We are evaluating the use of ZeroEQ, which is a C++ library that is specifically designed for a distributed, even-driven ecosystem [ZEROEQ].
[4] A higher storage tier level is assumed to provide access to data with higher performance.
[5] This challenge is not present in the BGAS framework, on which large parts of the design presented here is based, as all services are started at job start using the identity of the same user.

beyond the scope of SAGE as the project's focus is to explore the usability of the proposed architecture for the set of use cases defined in Section 3.

To ensure that resource usage of all runtime processes does not critically impact Mero services, the use of resources, such as CPU or memory must be isolated. The Linux cgroup feature will be used to enforce resource usage isolation [Linux-cgroup].

To provide a controlled and isolated environment for executing user applications as needed for the use case described in Section 3.2, the use of Linux Containers will be explored [Linux-containers]. Linux Containers is a technology that enables a set of processes to be isolated by means of virtualisation of kernel resources.

5.3. Semi-Persistent Data Cache Architecture

The semi-persistent cache is an architecture that will allow implementation of the use case described in Section 3.1 as follows:

1. A new workflow needs to be registered with the semi-persistent data cache service. This step involves providing a list of data products, which need to be processed, including their location in a parallel file system. Initially, all data products are likely in a low, i.e. slow, SAGE storage tier and are marked to be in the QUEUED state (see Figure 3 for a pictorial representation of the state machine).
2. The cache service will take care of creating a copy of the data products in the cache. The cache is located in a higher SAGE storage tier. To allow for multiple cache instances, the data products are marked as STAGING before a copy at a higher storage tier is started. Once the object is moved to a higher storage tier (or, alternatively, copy is available in that tier) the data product is marked STAGED.
3. A user application running on the compute nodes comprises a set of worker processes. When started or idle, a worker queries a designated cache instance for data products marked as STAGED.
4. During processing, secondary data products are created and written back to the cache. Once processing is completed, these secondary data products can be moved to the parallel file system.

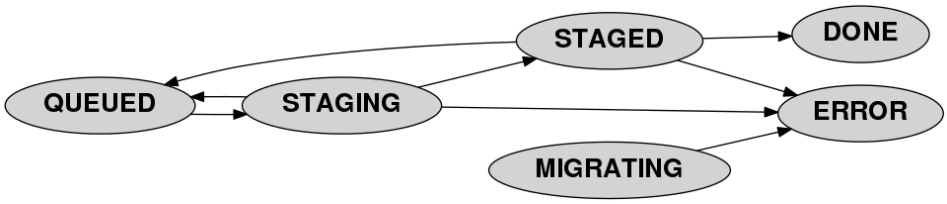

Figure 3: Semi-Persistent Cache Objects State Machine

The semi-persistent cache architecture comprises the following entities:

- **Cache Manager**: A daemon running on any SAGE node implementing a cache instance.
- **Cache Client Library**: A library of functions needed to use the semi-persistent cache service.
- **Cache Database**: A central database instance (e.g. Mero's key-value store) where, for each registered workflow, a list of data products including their state is stored and updated.
- **Cache Tools**: A set of tools needed to manage cache and database as well as data staging and migration.

Copies of data products in the cache exceed the lifetime of jobs and thus can be considered persistent. As the cache has, however, limited capacity, the cache manager is responsible for possible evictions of objects from the cache and thus the cache becomes semi-persistent.

5.4. Data Post-Processing Framework

The goal of the data post-processing framework is to enable execution of user code on SAGE nodes where the data is located. It relies on the ability of the management daemons to start user processes on the corresponding SAGE nodes. This concept of remote procedure execution is similar to what has been realised for Blue Gene Active Storage (BGAS). The main difference is that BGAS data locality could be managed in a simpler way as each compute node is connected to exactly one BGAS node.

The framework comprises the following components:

- Management daemon component that facilitates starting processes in user space.
- Client library providing routines for connecting to the management daemon.

The framework will provide the following functionality:

- Write a set of objects to SAGE.
- Start post-processing of these objects once the write is completed.
- Copy objects to archival storage (typically assumed to be file system based), if necessary.

For an initial prototype, it will be assumed that a common (parallel) file system is accessible from all SAGE nodes, such that the post-processing application can be referenced through a path in the file system. As the post-processing application may come with environmental dependences, the use of containers as a lightweight virtualisation approach will be explored.

The application will be started using a set of run-time arguments provided by the calling process. The framework will add the information required to identify the objects.

5.5. Virtual Memory Management Architecture

The virtual memory management architecture is implemented by a set of virtual memory manager daemons running on different SAGE nodes. Each cache manager is responsible to make a dedicated part of the overall data set accessible. Users connect to the daemons and send requests for data. When a copy of a requested data item is locally available as a cache block, it is immediately returned to the requestor. Otherwise, the data item is first loaded.

The virtual memory management architecture comprises the following components:

- **Virtual memory manager daemon**: Daemon running on a SAGE node with a Mero instance attached, which provides a data store for cache blocks. It processes incoming requests for data and checks whether it is responsible to provide the requested data. If yes, then it checks whether the data is available in the cache, otherwise it loads the data. Finally, the data is returned to the requester.
- **Virtual memory client library**: Library with functions used by a client application running, e.g. on a visualisation server, for communication with the virtual memory manager daemon.

Internally, the daemon is multi-threaded, with each thread running one of the following entities:

- **Event bus**: Central broker of all incoming requests and internal messages.
- **Virtual Memory Manager thread (VMMGR)**: The VMMGR manages the local data structures and cache.
- **Virtual Memory Worker thread (VMWRK)**: One or more VMWRK threads load data items into the local cache on request by the VMMGR. They may also pre-process data items upon loading.
- **Virtual Memory Communicator thread (VMCOM)**: The VMCOM forwards cache blocks to the requestor once they are available in the local DRAM cache.

See
Figure 4 for an overview on the architecture of the virtual memory manager daemon.

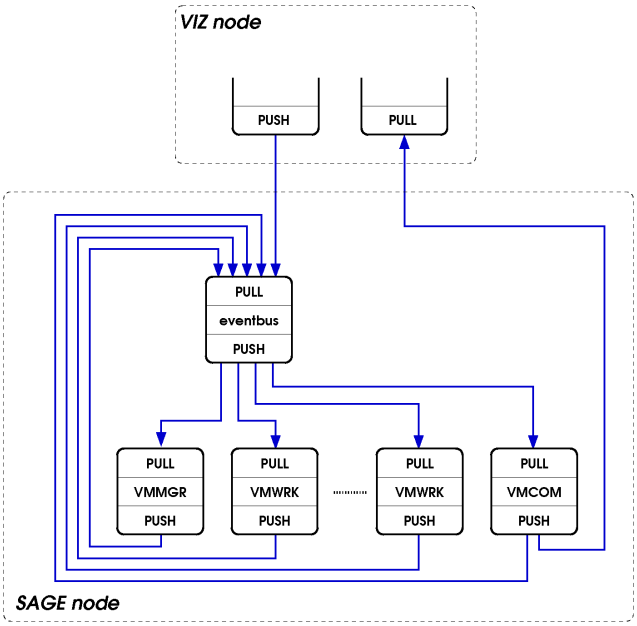

Figure 4: Architecture of the Virtual Memory Manager Daemon.

6. Implementation Plan

The following landmarks guide implementation of the architecture outlined in this document:

1. Full design specification written and internal review completed.
2. Initial implementation of base services, running in virtual machine, using multiple instances of Mero, and functional tests completed.
3. Synthetic performance testing of base services completed, using a hardware deployment of Mero.
4. Initial implementation of the semi-persistent cache infrastructure, running in virtual machine and using one cache instance, and functional tests completed.
5. Initial implementation of the data post-processing framework, running in virtual machine using multiple Mero instances, and functional tests completed.
6. Initial implementation of the virtual memory management architecture, running in virtual machine using one Mero instance, and functional tests completed.[6]
7. Internal review of architecture implementation completed and, if needed, architectural changes defined.
8. Performance evaluation of the semi-persistent data cache completed and tested for JURASSIC.[7]
9. Performance evaluation of the data post-processing framework completed for realistic workloads using NEST. [8]
10. Performance evaluation of the virtual memory hierarchy for a volume rendering visualisation application completed. [9]

[6] This work will be performed as part of task T4.3 (“Visualisation”).
[7] This work will partially be performed as part of tasks T1.3 (“Application re-design, porting and optimization”) and T5.3 (“Evaluation of applications on SAGE prototype & Final Evaluation Report”).
[8] See previous footnote.
[9] This work will partially be performed as part of tasks T4.3 (“Visualisation”) and T5.3 (“Evaluation of applications on SAGE prototype & Final Evaluation Report”).

7. Test and Evaluation Plan

In this section, we provide a list of function and performance tests, which we plan to set up while implementing the runtime infrastructure. Furthermore, we provide an overview of specific aspects of the described use cases that we plan to evaluate.

7.1. Functional Tests for Base Services

- Start-up of management daemons and registration of all management daemons.
- Testing of identity control mechanisms.
- Validation of correct discovery of locally available resources.
- Communication of clients with the management daemon at both low and maximally high message rates to check for protocol robustness.
- Spawning of processes using a different user ID and verification of resource isolation mechanisms.
- Validation of the logging infrastructure for different configurations.

7.2. Performance Tests

- Message rates for different input patterns (e.g. a different number of clients).
- Measurement of message handling latency.
- Measurement of overhead for process spawning.

7.3. Use Case Evaluation

In this section we describe how we plan to evaluate the run-time system for different use cases.

7.3.1. Semi-Persistent Cache Architecture

The semi-persistent cache will be evaluated using real life workflows of the JURASSIC application. Performance metrics that will be evaluated include,, but are not limited to the following:

- Average time per data object that a user spent waiting for a data object to become available.
- Average cache utilization and number of cache evictions.

Furthermore, we plan to explore the following architectural aspects that are expected to have significant performance impacts:

- Performance as a function of the cache capacity (including the extreme case of zero cache size).
- Performance impact of different staging and cache eviction policies and settings.
- Performance impact of different object placement strategies in the scenario of multiple cache instances.

7.3.2. Data Post-Processing Framework

Functional testing of the post-processing data infrastructure includes the following:

- Spawning of remote procedure and verification of return codes.
- Verifying the application of remote procedures to objects.
- Verifying resource isolation mechanisms.

The post-processing data framework will be evaluated using realistic use cases based on the neuronal network simulator (NEST) or similar applications. Performance metrics that will be evaluated include, but are not limited to, the following:

- Measurement of overhead to spawn remote procedures for different workloads.
- Performance, as a function of different object placement strategies, for a select set of workloads.

7.3.3. Virtual Memory Hierarchy Infrastructure

Functional and synthetic performance testing of the virtual memory hierarchy infrastructure includes the following:

- Correct completion of data requests at single and high request rates.
- Response time for data requests and effective data bandwidth for different request patterns.

The semi-persistent cache will be evaluated using realistic use cases for volume rendering-based visualisation. Performance metrics that will be evaluated include, but are not limited to, the following:

- Effective bandwidth for data transfer between SAGE nodes and the visualisation unit as function of the number of used nodes.
- Cache hit rates for different access patterns and data distribution strategies.

8. References

[Aqua] http://science.nasa.gov/missions/aqua/ (accessed 23.07.2016).

[Fogal2010] T. Fogal and J. Krüger, “Tuvok - An Architecture for Large Scale Volume Rendering,” 15th Vision, Modeling and Visualization Workshop ’10, pp. 139–146, 2010.

[Hadwiger2012] M. Hadwiger, J. Beyer, W. K. Jeong, and H. Pfister, “Interactive volume exploration of petascale microscopy data streams using a visualization-driven virtual memory approach,” IEEE Transactions on Visualization and Computer Graphics, 18(12):2285–2294, 2012.

[JURASSIC] http://www.fz-juelich.de/ias/jsc/EN/AboutUs/Organisation/ComputationalScience/Simlabs/slcs/JURASSIC/_node.html

[Linux-cgroup] http://man7.org/linux/man-pages/man7/cgroups.7.html

[Linux-containers] https://linuxcontainers.org/

[Linux-exec] http://man7.org/linux/man-pages/man3/exec.3.html

[NERSC2016] http://www.nersc.gov/users/computational-systems/cori/burst-buffer/ (accessed 04.07.2016).

[NEST] http://www.nest-simulator.org/

[NEST2007] M.-O. Gewaltig and M. Diesmann, „NEST (NEural Simulation Tool),” Scholarpedia, 2007 (http://www.scholarpedia.org/article/NEST_%28NEural_Simulation_Tool%29).

[SAGE2016] SAGE, "Documentation of Application Requirements”, deliverable D1.1, May 2016.

[SASL] https://tools.ietf.org/html/rfc4422

[ZEROEQ] https://github.com/HBPVIS/ZeroEQ

[ZERMQ] http://zeromq.org/

