Design of a Hierarchical Software-Defined Storage System for Data-Intensive Multi-Tenant Cloud Applications

Pieter-Jan Maenhaut*†, Hendrik Moens†, Bruno Volckaert†, Veerle Ongenae* and Filip De Turck†
*Ghent University, Faculty of Engineering and Architecture, Dept. of Industrial Technology and Construction
Valentin Vaerwyckweg 1, 9000 Ghent, Belgium
†iMinds – INTEC, Ghent University, Dept. of Information Technology
Gaston Crommenlaan 8 bus 201, 9050 Ghent, Belgium
Email: pieterjan.maenhaut@intec.ugent.be

***Abstract*—Software-Defined Storage (SDS) is an evolving concept in which the management and provisioning of data storage is decoupled from the physical storage hardware. Data-intensive multi-tenant SaaS applications running on the public cloud could benefit from the concepts introduced by SDS by managing the allocation of tenant data from the tenant's perspective, taking custom tenant policies and preferences into account.**

In this paper, we propose the design of a scalable multi-tenant SDS system. In our approach, tenants are hierarchically clustered based on multiple scenario-specific characteristics. The storage elasticity component of the SDS system is responsible for the dynamic (re-)allocation of tenant data over the available storage resources. It invokes the Hierarchical Bin Packing algorithm introduced in this paper to determine an optimized distribution of tenant data based on the hierarchical tenant tree.

We evaluate our system by means of two case studies based on real-life data sets. Experiments confirm that the Hierarchical Bin Packing algorithm achieves a good performance, with execution times below 100 ms to calculate the allocation for 1000 tenants in a worst-case scenario. Furthermore, our system achieves an average utilization of the storage resources close to the configured allocation factor, with reallocation of tenant data balanced over time.

I. Introduction

Software-Defined Storage (SDS) [1] is an evolving concept for the management of data storage from the software's perspective, independent of the underlying hardware. A SDS system manages the policy-based provisioning of data storage, and virtualization is often used to provision the required resources, similar as with cloud computing.

With public cloud computing, infrastructure providers usually apply a cloud pricing model in which the customer is charged based on the actual resource usage (pay-as-you-go pricing model) [2]. Multi-tenancy [3] enables the sharing of resources by multiple client organizations, referred to as tenants. As a result, adding multi-tenancy to a cloud application increases the utilization of available hardware resources, resulting in lower overall application costs. Although a single instance is shared between multiple tenants, the instance needs to behave like a private instance towards every tenant by guaranteeing both data separation and performance isolation. A scalable multi-tenant application should also be able to react to sudden changes in demand, by provisioning the optimal amount of resources in order to handle the current load, and distributing the different tenants over the available instances.

For data-intensive multi-tenant applications, every tenant will have a certain amount of persistent data which can be distributed over multiple storage volumes. As a result, a scalable SDS system is required whose task is to provision the optimal amount of storage resources, and to distribute the tenant data among the different available volumes, but guaranteeing a clear isolation of tenant data.

In this paper, we propose the design of a multi-tenant software-defined storage system. The system invokes the Hierarchical Bin Packing (HBP) algorithm to determine an optimized allocation of tenant data over multiple logical disks. The remainder of this paper is structured as follows. In the next section we will discuss related work. Afterwards, in Section III we will explain the need for hierarchical clustering of the tenants. In Section IV we present the architecture of our system and introduce the Hierarchical Bin Packing Algorithm in Section V. In Section VI, we will present some of our evaluation results together with a discussion in Section VII and in Section VIII we state our conclusions and discuss avenues for future research.

II. Related Work

In previous work [4], [5], we worked on the design of a data management framework which can be used to extend existing multi-tenant cloud applications in order to achieve high scalability of the database layer. This database layer consists of multiple relational databases, and the framework manages the distribution and retrieval of tenant data over the available instances, but guaranteeing the correct functioning of the data queries. In this paper, we focus on the design of a SDS system for managing the allocation of blob storage instead of relational data. This leads to a very different approach for the allocation of tenant data, with a strong focus on data isolation, and the design of a new algorithm.

Ceph [6] is a distributed object store and file system designed to provide excellent performance, reliability and

978-3-901882-77-7 © 2015 IFIP

scalability. Ceph stores client data as objects within storage pools, and uses the CRUSH [7] algorithm to allocate the objects over placement groups. The CRUSH algorithm also uses a hierarchical structure, called the hierarchical cluster map, but this hierarchical structure is used to manage the storage devices, which are the leaves of the tree. The main difference with our solution is that CRUSH will distribute data using a pseudo-random function. Each object is mapped to a list of devices on which to store object replica, approximating a uniform probability distribution. We however want to distribute data based on tenant-specific parameters, where every tenant is mapped to a single storage device (if possible). Our solution will not necessarily lead to a uniform distribution of data, but it will try to maximise isolation of tenant data.

In [8] the performance of a data cluster based on the Ceph platform with geographically separated nodes is evaluated. The authors focus on high availability, by allocating copies of the data over multiple distributed storage nodes, and measuring the required bandwidth. We however focus on the distribution of tenant data based on selected characteristics, and less on the high availability of data. The solution proposed in this paper could however be used to extend our system to support distributed high-availability.

III. Hierarchical Clustering of Tenants

In the presented approach, the different tenants are hierarchically organized using a tree structure. There are several reasons to do so. First of all, multi-tenant applications are often used by multiple organizations, the tenants. As large organizations tend to consist of multiple independent divisions, this already introduces the need for subtenants or even sub-subtenants. But there might also be other reasons to structure tenants hierarchically. For example when the tenants using the application are geographically distributed, it might be a good idea to cluster tenants based on their location, with the location as a virtual parent node and the tenants as child nodes. Tenants could also be clustered based on other characteristics, such as the selected Service-Level Agreement (SLA), with different nodes for the different types of SLAs. In general, tenants can be clustered based on multiple characteristics, with the most significant characteristics at the highest level of the tree structure. As a result the hierarchical tree structure, which we refer to as the tenant tree, will have multiple levels, as illustrated in Figure 1, and more levels in the tenant tree will result in better results for the HBP algorithm.

IV. Architecture Overview

Figure 2 illustrates the concept of a scalable system for data-intensive multi-tenant cloud applications. For data storage, a Logical Unit Number (LUN) is often used to identify a logical unit that can be addressed by the SCSI protocol or Storage Area Network (SAN) protocols [9]. Although the term LUN refers to the number of the logical disk, in this paper we will use the term to refer to the logical disk itself, as is often done in literature. When an authenticated tenant user wants to access the multi-tenant application, he first connects to the load balancer (1) in order to select one of the available application server instances (2). In order to retrieve the application data, the application server connects to the storage pool where the tenant data is stored (3).

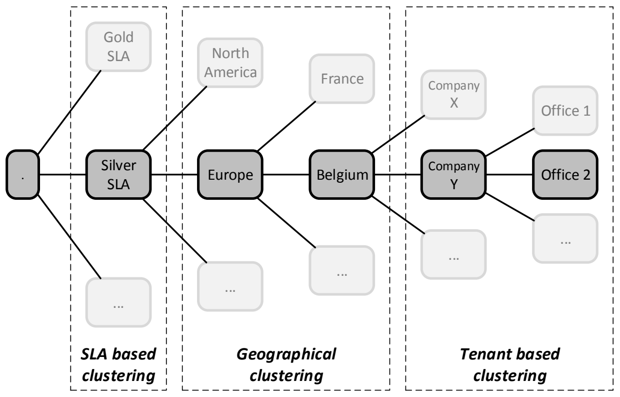

Fig. 1. Example tenant tree where tenants are clustered based on multiple characteristics, with more significant characteristics such as the selected SLA at a higher level in the tree structure.

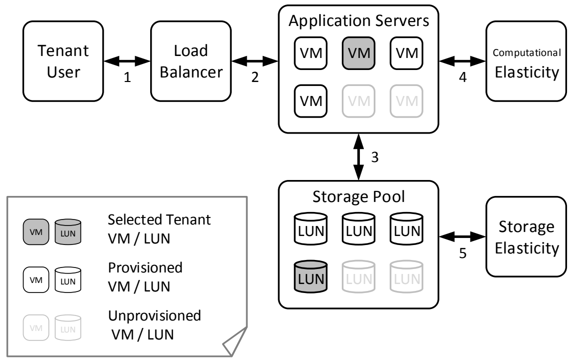

Fig. 2. General overview of a scalable system for data-intensive multi-tenant cloud applications.

To achieve high scalability of the computational resources, the computational elasticity component monitors the current load on the provisioned application server instances (4). As the load increases, additional instances will be provisioned. To achieve high scalability of the storage resources, the storage elasticity component monitors the current usage of the provisioned LUNs (5). When a single LUN reaches a certain threshold, the reallocation factor, tenant data is reallocated and an additional LUN can be provisioned. Similarly, if the current load on either the application servers or the current usage on the provisioned LUNs decreases significantly, one or more application servers or LUNs should be de-provisioned, requiring the reallocation of some of the tenants.

The application servers consist of multiple Virtual Machine (VM) instances, and on every instance the multi-tenant application is running on top of a Multi-Tenant Datastore Selection (MTDS) module, responsible for communicating with the storage pool. This module has access to the current allocation scheme in order to select the correct LUN, and this scheme is updated by the storage elasticity component whenever tenant data is reallocated. The main task of the MTDS module is to select the required LUN, and to verify if the current tenant user has the required permissions to read and/or modify the selected tenant data.

The storage pool consists of a set of provisioned LUNs. The number of LUNs can change over time, and every LUN can hold data belonging to multiple (sub)tenants. Data belonging to a single (sub)tenant is however is always allocated to a single LUN, unless it is too big in size. The LUN can be replicated to achieve high availability. The set of LUNs is assumed to be homogeneous, meaning that they have identical characteristics, such as the size and format. An extension for a heterogeneous storage pool is possible but is out of scope for this paper.

The storage elasticity component handles the dynamic behaviour of the storage pool. It invokes the HBP algorithm to find a feasible allocation over the available LUNs, and (re)allocates the tenant data.

V. Hierarchical Bin Packing

The storage elasticity component invokes the HBP algorithm to find a feasible allocation of tenant data over multiple bins (the LUNs). Three configurable parameters are used by this component:

- $SIZE$, the maximum size of a single bin
- AF, the allocation factor $(0 < AF < 1)$
- RF, the reallocation factor $(RF > AF, 0 < RF < 1)$

When the algorithm is invoked, it will allocate the tenant data over multiple bins with the following constraint:

$$bin.currentUsage <= AF \times SIZE$$

In the above equation, $bin.currentUsage$ denotes the current usage of a single LUN. As the amount of tenant data grows over time, reallocation of tenant data might be required. More specific, the algorithm will be reinvoked and tenant data will be reallocated whenever one of the LUNs violates the following constraint:

$$bin.currentUsage <= RF \times SIZE$$

The algorithm takes a set of tenants, together with their current size as input, and returns a mapping from tenants to multiple bins. The tenants are hierarchically organized using a tree structure, the tenant tree, in which each node represents a (sub)tenant. A single node is represented by the following structure:

```
class Node{
  String name;       // tenant identifier
  Node[] children;   // pointer to child nodes
  double nodeSize;   // size of current node
  double treeSize;   // size of (sub)tree
                // with current node as root
}
```

The pseudo-code of the HBP algorithm is presented in Algorithm 1. The algorithm takes two input parameters, a pointer to the root node and an empty set of bins (representing

TABLE I
Summary of used symbols

Symbol	Description
$SIZE$	Maximum size of a single bin
AF	Allocation factor
RF	Reallocation factor
$root$	Pointer to root of tenant (sub)tree
$n.nodeSize$	Size of node n (without children)
$n.treeSize$	Total size of (sub)tree with n as root
$b.currentUsage$	Current usage of bin b

Algorithm 1 The Hierarchical Bin Packing (HBP) algorithm

```
Input: Pointer to root node ($root$), set of LUNs ($bins$)
Output: Set of LUNs ($bins$)
  if $root.treeSize \leq AF \times SIZE$ then
    Allocate whole (sub)tree with size $root.treeSize$ to a
    single bin using the First-Fit Bin Packing algorithm
  else
    $sorted$ = sort($\{root.nodeSize\} \cup \{n.treeSize$ for $n$ in
    $root.children\}$)
    for $n$ in $sorted$ do
      if $n$ is $root$ then
        Allocate root node only with size $root.nodeSize$ to
        a single bin using the First-Fit Bin Packing algorithm
      else
        Invoke algorithm with $n$ as $root$ and $bins$ as input
        parameters (recursive call)
      end if
    end for
  end if
  return $bins$
```

the LUNs), and returns this set with all nodes allocated. The second input parameter is required for the recursive calls in the algorithm. For completeness, Table I provides a summary of the used symbols.

If the whole (sub)tree can be allocated to a single bin, the algorithm will do so. For allocating a node or (sub)tree, the First-Fit Bin Packing (FFBP) algorithm is used, meaning that the amount of data is allocated to the first bin with enough space left, or to a new bin if no other bins are suitable. Note however that a (sub)tree will not be split, the whole (sub)tree will be allocated to a single suitable bin. If the (sub)tree does not fit a single bin, the algorithm will be invoked recursively. First, all child subtrees are sorted based on their $treeSize$, together with the current root node based on its $nodeSize$. Next, for every subtree in this sorted set, the algorithm is invoked with a pointer to the subtree as root. The root node itself is allocated using the same FFBP algorithm as mentioned above. In fact any bin packing algorithm can be used for the implementation, but we selected the FFBP algorithm as this approximation algorithm already provides acceptable results with very low execution times.

The HBP algorithm works fine if all nodes have a $nodeSize \leq AF \times SIZE$. In some cases however, there might exist some bigger nodes which don't fit a single bin. As a result, the tenant tree is first 'fixed' before invoking the

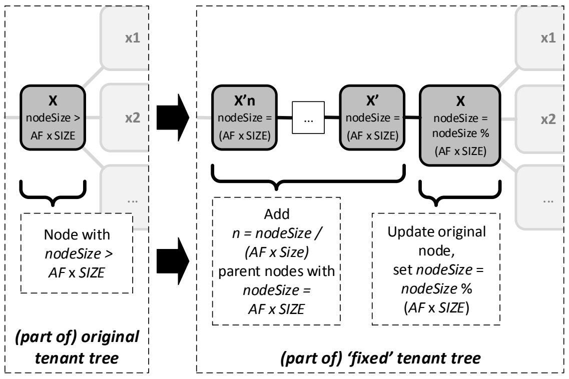
x1
x2
...
X
nodeSize > AF x SIZE
Node with nodeSize > AF x SIZE
(part of) original tenant tree
X'n
nodeSize = (AF x SIZE)
...
X'
nodeSize = (AF x SIZE)
X
nodeSize = nodeSize % (AF x SIZE)
Add n = nodeSize / (AF x Size) parent nodes with nodeSize = AF x SIZE
Update original node, set nodeSize = nodeSize % (AF x SIZE)
(part of) 'fixed' tenant tree

Fig. 3. Fixing the tenant tree before invoking the HBP algorithm.

algorithm. During this pre-processing step, big nodes are split into multiple smaller nodes as illustrated in Figure 3.

VI. Evaluation Results

All experiments were executed on a Linux server with an Intel Core i5 CPU (1.40 GHz) and 4 GiB of 1600MHz DDR3 memory. The HBP and FFBP algorithms were implemented in C++. For the implementation of the FFBP algorithm, the First Fit Decreasing strategy was used, meaning that the algorithm first sorts the items to be inserted by their sizes in decreasing order, and then inserts each item into the first bin with sufficient remaining space.

A. Evaluation of efficiency

In order to evaluate the efficiency of the system, we focused on 3 different metrics. The first metric is the average LUN utilization, as a higher value for this metric leads to fewer LUNs and hence lower operating costs. The second metric focuses on the reallocations, and we look at both the number of reallocations as the amount of data to migrate. The last metric is the average distance between the nodes allocated to a LUN, as a lower value indicates a better clustering.

For our simulations, we worked with 2 case studies based on real-life data sets. The first case study is the implementation of a population register, in which for every inhabitant a small amount of data is stored. The data set is based on the yearly population for every town in Flanders over the period 2005 to 2012. These numbers are available from the official website of Flanders [10]. The different towns (tenants) were geographically clustered based on the capital city, region and province to create the hierarchical tenant tree. Including the internal nodes, the whole tenant tree for this scenario consists of 946 nodes. This data set is relevant because it is a good example of a large slow-growing data set.

We experimented with different values for the configurable parameters $SIZE$, AF and RF, but most experiments lead to similar results. Note that for this scenario, the parameter $SIZE$ is used to denote the maximum number of inhabitants that can be stored on a single LUN. Figure 4 illustrates the average LUN utilization over the different years for both the HBP algorithm and the FFBP algorithm for one of our simulations. No reallocations were required over the years, as

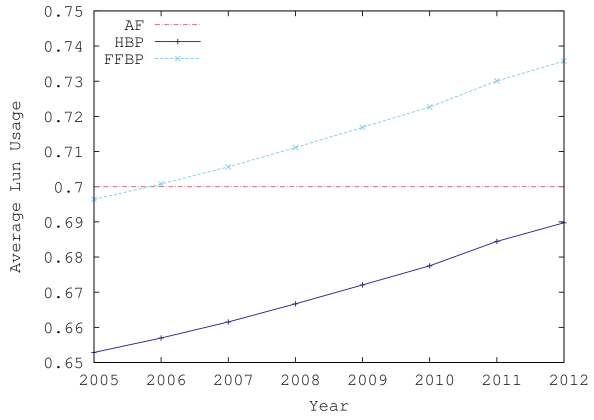

Fig. 4. Average LUN utilization for the slow-growing data set over the different years with $SIZE = 10^6$, $AF = 0.7$ and $RF = 0.9$.

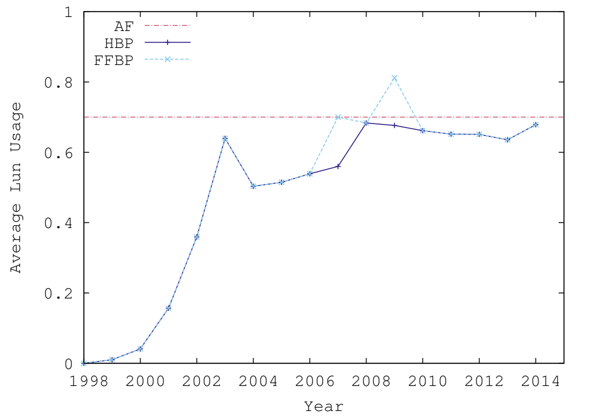

Fig. 5. Average LUN utilization for the fast-growing data set over the different years with $SIZE = 10^8$, $AF = 0.7$ and $RF = 0.9$.

expected for a slow-growing data set. Furthermore, the FFBP algorithm initially achieves a slightly higher LUN utilization than the HBP algorithm, as the FFBP allocated the data over 15 LUNs whereas the HBP algorithm required 16 LUNs. The additional cost for the latter however is strongly compensated by the fact that the data on the LUNs is geographically clustered, which is not at all the case for the FFBP algorithm.

Our second case study is similar as the first one, but the data set is based on the number of fixed broadband internet subscribers per country worldwide over the last 17 years, which is a large fast-growing data set. This data is available from the World Bank Open Data [11]. The different countries (tenants) are hierarchically clustered based on their world region, resulting in a tenant tree with a total of 222 nodes. We did similar experiments as with the previous scenario, but for this scenario we paid extra attention to the migration size (the percentage of data that needs to be migrated between LUNs between 2 consecutive allocations), as we expect a lot of reallocations due to the fast-growing behaviour.

Figure 5 and Figure 6 illustrate the average LUN utilization and the migration size for one of our experiments, whereas Table II provides a detailed overview of the results. As can be seen from these results, the first 5 years the amount of data

TABLE II
SIMULATION RESULTS FOR THE FAST-GROWING DATA SET OVER THE DIFFERENT YEARS WITH $SIZE = 10^8$, $AF = 0.7$ AND $RF = 0.9$.

	HBP			FFBP			
Year	Avg. Utilization (%)	# LUNs	Migr. Size (%)	Avg. Utilization (%)	# LUNs	Migr. Size (%)	Total Size
1999	1.013788	1	-	1.013788	1	-	1013788
2000	4.083473	1	0	4.083473	1	0	4083473
2001	15.661822	1	0	15.661822	1	0	15661822
2002	35.936250	1	0	35.936250	1	0	35936250
2003	63.967070	1	0	63.967070	1	0	63967070
2004	50.343976	2	32.0369	50.343976	2	30.4783	100687952
2005	51.455433	3	28.0221	51.455433	3	38.1982	154366301
2006	53.902831	4	10.1821	53.902831	4	54.5252	215611325
2007	55.996213	5	35.9997	69.995267	4	44.1721	279981069
2008	68.334749	5	31.0423	68.334749	5	31.425	341673747
2009	67.650500	6	29.8826	81.180600	5	0	405903003
2010	66.163573	7	27.6989	66.163573	7	38.9445	463145017
2011	65.160919	8	34.1534	65.160919	8	36.0545	521287355
2012	65.088807	9	24.9946	65.088807	9	22.7164	585799265
2013	63.551126	10	31.3191	63.551126	10	31.0177	635511265
2014	67.862235	10	0	67.862235	10	0	678622358

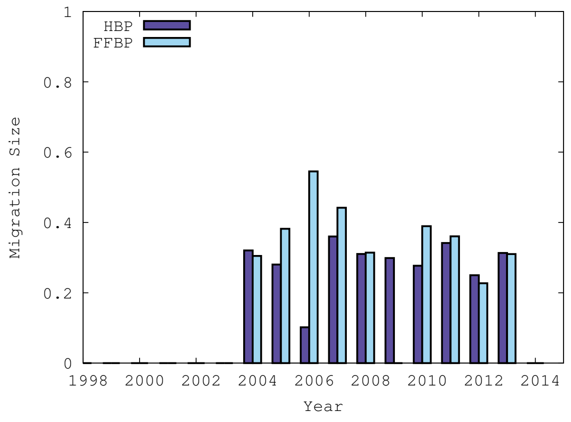

Fig. 6. Percentage of data to reallocate for the fast-growing data set over the different years with $SIZE = 10^8$, $AF = 0.7$ and $RF = 0.9$.

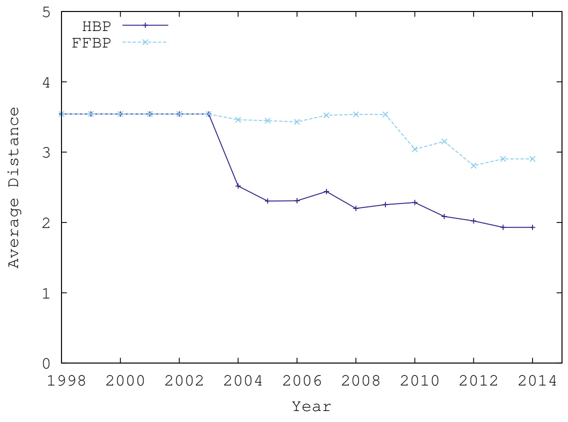

Fig. 7. Average distance over the different bins for the fast-growing data set over the different years with $SIZE = 10^8$, $AF = 0.7$ and $RF = 0.9$.

starts growing, resulting in no reallocations as all data can still be allocated to a single LUN, but from 2004 on there is an enormous grow in size, leading to reallocations almost every year. This graph clearly shows the main difference between the two algorithms. The FFBP algorithm tends to reach a higher average LUN utilization, but this results in very large migration sizes (almost 55% of data is reallocated between 2005 and 2006). The HBP algorithm reaches a slightly lower average LUN utilization, because of the forced clustering, but this results in more balanced migration sizes and an almost constant average utilization which is still very close to the selected AF.

We also measured the average distances over the different LUNs. To achieve this, for every LUN we measured the distance between all nodes allocated to this bin. The distance between two nodes is defined as the number of edges between the two nodes in the tenant tree, following the shortest path. Figure 7 gives an overview of the measured average distances over the different years. During the first years, both algorithms have the same average distance, which is expected as both algorithms allocate all tenants to a single LUN. However, as soon as the amount of data becomes too large to allocate to a single LUN, the HBP algorithm achieves a much lower distance than the FFBP algorithm, indicating that the algorithm is able to cluster related tenants.

B. *Worst-case performance*

In order to evaluate the performance of the HBP algorithm, we focused on the worst-case scenario. The worst-case scenario occurs when the tenant tree is a linear graph, meaning that there is only 1 leaf node connected to the root node (with possible multiple levels in between). We refer to this scenario as *vertical growth*. Especially when every node in the tree is large, meaning that no nodes can be clustered together on a single LUN, the algorithm will do the maximum number of recursive calls. In theory, we expect that the algorithm will still achieve the same complexity as the FFBP algorithm, $O(n \log n)$, which will be confirmed by our experiments.

To simulate this behaviour, we evaluated a scenario in which we started with a single node in the tenant tree, which is both root and leaf, with a maximum $nodeSize = AF \times SIZE$.

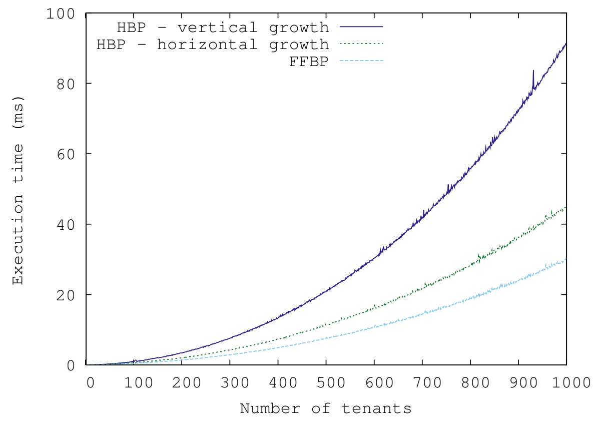

Fig. 8. Execution times of the HBP algorithm versus the FFBP algorithm for both worst-case scenarios.

Every iteration, a new child node n was added to the leaf node of the tree with $n.nodeSize = n.parent.nodeSize - 10^{-6}$. This results after i iterations in a tenant tree with 1 leaf node, $i-1$ internal nodes in between the leaf node and the original root node, and $i+1$ levels in the tree. Every iteration was repeated 100 times in order to measure the average execution time.

We also executed this experiment with a second worst-case scenario. In this scenario, there is a single root with an increasing number of child nodes (tenants) over the different iterations, and we will refer to this scenario as *horizontal growth*. Every child node was assigned a maximum $nodeSize = AF \times SIZE$ to prevent clustering.

Figure 8 illustrates the average execution times for the HBP algorithm and the FFBP algorithm over the different iterations for both scenarios. As can be seen from this figure, for the *vertical growth* the HBP algorithm has slightly higher execution times than the FFBP algorithm, due to the overhead of the recursive calls and the growing call stack. However, as it still takes less than 100 ms to calculate the allocation for 1000 tenants, this is not really an issue. For the second scenario, the *horizontal growth*, the overhead is much smaller, because there is only 1 recursive call active at any time (for the current subtenant), and the call stack is not growing. Note that for both scenarios the FFBP algorithm achieves similar execution times as the hierarchical structure has no influence on the performance of this algorithm.

VII. Discussion

In the previous section, we evaluated our system by analysing both the efficiency and the performance of the HBP algorithm, which is the core of our system. Compared to the FFBP algorithm, the HBP algorithm achieves a slightly lower LUN utilization, but the average is still close to the selected allocation factor AF. The main benefits from the algorithm are the forced clustering of tenants, resulting in lower LUN distances, and faster provisioning of additional storage resources, resulting in a more balanced reallocation of tenant data over time. Furthermore, the algorithm has very low execution times even for a worst-case scenario.

In Figure 2, any load balancer mechanism could be used for implementing the load balancer for the application servers. We however prefer to use tenant-based load balancing, in which users belonging to the same tenant are always routed to the same application server instance, and the HBP algorithm could even be used for implementing the load balancer. By doing so, multiple tenants can share the same instance if the tenants are small. There are several reasons for this approach. First of all, it improves the performance isolation. If a tenant puts a heavy load on the application, only a single application instance is affected, minimizing the impact for other tenants. Next, if the tenants are physically distributed, an instance can be selected that is close to the tenant's location. Finally, and this might be one of the most important reasons, locking and synchronisation of data becomes easier as there is a one-to-one mapping from the application server instance to the tenant data located in the storage pool.

The multi-tenant SDS system presented in this paper can be implemented on any elastic cloud system. An implementation on a hybrid cloud in particular could however be very interesting. In this case, the public cloud can be used for provisioning the application servers, whereas all tenant data can be stored on a geographically distributed private storage cloud. Using a private cloud for the storage provides enough flexibility to meet legal and business data archival requirements for the tenant data, as compliance with regulatory policies on data remains a key hurdle to cloud computing [12].

VIII. Conclusions and Future Work

In this paper, we presented a software-defined storage system for data-intensive multi-tenant applications. On every application server instance the multi-tenant application is running on top of a MTDS module, responsible for communication with the storage pool. In our approach, the tenants are hierarchically organized using a tree structure, the tenant tree, and the storage elasticity component invokes the HBP algorithm to determine an optimized allocation of tenant data over the different storage resources.

The implementation of the HBP algorithm has very low execution times even for the worst-case scenario. Compared to the FFBP algorithm, the HBP algorithm achieves a slightly lower LUN utilization, but the average is still close to the selected allocation factor. On the other hand, as data is clustered based on selected attributes, the reallocation of tenant data is more balanced compared to the FFBP algorithm, and the algorithm will achieve a much better isolation of tenant data as related tenants are allocated to a single LUN.

In future work, we will investigate techniques to further reduce the migration size. The evaluation results presented in this paper however confirm that the presented system could provide a solid basis for implementing a multi-tenant SDS system.

Acknowledgment

This research is partly funded by the IWT SBO DeCoMAdS project.

References

[1] M. Carlson, A. Yoder, L. Schoeb, D. Deel, and C. Prattr, “Software defined storage,” SNIA, Tech. Rep., Mar. 2014. [Online]. Available: http://www.snia.org/sds

[2] M. Armbrust, R. Fox, Armandoand Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, “Above the clouds : A Berkeley view of cloud computing,” University of California at Berkley, Tech. Rep., 2009.

[3] C. J. Guo, W. Sun, Y. Huang, Z. H. Wang, and B. Gao, “A framework for native multi-tenancy application development and management,” in *9th IEEE International Conference on E-Commerce and the 4th IEEE International Conference on Enterprise Computing, E-Commerce, and E-Services, 2007. CEC/EEE 2007.*, Tokyo, Japan, Jul. 2007, pp. 551 – 558.

[4] P.-J. Maenhaut, H. Moens, V. Ongenae, and F. D. Turck, “Scalable user data management in multi-tenant cloud environments,” in *Proceedings of the 10th International Conference on Network and Service Management 2014 (CNSM2014)*, Rio de Janeiro, Brazil, Nov. 2014, pp. 268 – 271.

[5] ——, “Design and evaluation of a hierarchical multi-tenant data management framework for cloud applications,” in *The Seventh IFIP/IEEE International Workshop on Management of the Future Internet (ManFI2015)*, Ottawa, Canada, May 2015, pp. 1208 – 1213.

[6] S. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn, “Ceph: A scalable, high-performance distributed file system,” in *Proceedings of the 7th Conference on Operating Systems Design and Implementation (OSDI ’06)*, Nov. 2006.

[7] S. A. Weil, S. A. Brandt, E. L. Miller, and C. Maltzahn, “Crush: Controlled, scalable, decentralized placement of replicated data,” in *Proceedings of the 2006 ACM/IEEE Conference on SuperComputing (SC ’06)*, 2006.

[8] D. Malanik and R. Jaek, “The performance of the data-cluster based on the ceph platform with geographically separated nodes,” in *Mathematics and Computers in Sciences and in Industry (MCSI), 2014 International Conference on*, Sept 2014, pp. 299–307.

[9] J. Long, *Storage Networking Protocol Fundamentals*. Cisco Press, 2005.

[10] FLANDERS.be - the official website of Flanders. [Online]. Available: http://www.flanders.be/en

[11] World Bank - Open Data. [Online]. Available: http://data.worldbank.org

[12] M. Henze, M. Grossfengels, M. Koprowski, and K. Wehrle, “Towards data handling requirements-aware cloud computing,” in *Proceedings of the 2013 IEEE 5th International Conference on Cloud Computing Technology and Science (CloudCom)*, vol. 2, Dec. 2013, pp. 266–269.