Towards Resource Disaggregation - Memory Scavenging for Scientific Workloads

Alexandru Uta*†, Ana-Maria Oprescu*†, Thilo Kielmann*†
*Computer Science Institute, Vrije Universiteit Amsterdam, The Netherlands
†Amsterdam Department of Informatics, The Netherlands
{a.uta, a.m.oprescu, thilo.kielmann}@vu.nl

***Abstract*—Compute clusters, consisting of many, uniformly built nodes, are used to run a large spectrum of different workloads, like tightly coupled (MPI) jobs, MapReduce, or graph-processing data-analytics applications, each of which with their own resource requirements. Many studies consistently highlight two types of under-utilized cluster resources: memory (up to 50%) and network. In this work, we take a step towards (software) resource disaggregation, and therefore increased resource utilization, by designing a memory scavenging technique that makes unused memory available to applications on other cluster nodes. We implement this technique in MemFSS, an in-memory distributed file system. The scavenging MemFSS extends its storage space by taking advantage of the unused memory and bandwidth of cluster nodes already running other tenants' applications. Our experiments show that our memory scavenging approach incurs negligible overhead (below 10%) for most tenant applications, while the compute resource comsumption of MemFSS applications is largely reduced (by 17%-74%).**

I. Introduction

For decades (commodity) clusters have provided the computing infrastructure needed by a large variety of applications and processing frameworks. Such computations range from tightly coupled high-performance computing simulations to data-driven analytics tasks. Therefore, clusters have evolved to support a multitude of applications with different characteristics: several computing platforms rely on high-bandwidth, low-latency network interconnects, while others rely on large and efficient storage solutions. The common feature of typical cluster applications is that they are optimized to maximize the usage of processing units (CPUs).

Hence, the most utilized resource in clusters is the CPU. However, there are two other resources that are heavily under-utilized in clusters: *memory* and *network (bandwidth)*. While in the network case this situation was only recently introduced by the steady increase in network performance, memory under-utilization has been reported since the early 2000s [1], [2] and it is still the case [3], [4], as memory sizes per machine have largely increased.

Our previous work [5], [6] takes advantage of *memory* and *fast networks* (InfiniBand, 10G Ethernet) to accelerate the I/O of scientific workflows by means of an in-memory distributed file system. Such applications are composed of many (typically short) tasks that communicate by means of files, generating large amounts of intermediate data (hundreds of GBs to TBs) at runtime. As the availble memory per node is much smaller than the required disk space, an application would typically require a larger number of nodes to accommodate its data footprint (the maximum span of intermediate data). At the same time, many scientific workflows suffer from poor computational scalability due to their inherent structure: large parallel stages of tasks are followed by long-running data-aggregation or partitioning stages. Therefore, during those stages which exhibit less achievable parallelism, (most of) the CPUs of the reserved cluster nodes are under-utilized.

We observe two complementary resource utilization patterns. First, typical cluster applications exhibit a high CPU utilization while leaving large amounts of memory and network under-utilized. Second, our in-memory distributed file system highly utilizes memory and network, but due to the workflow structure, the CPUs are under-utilized. Therefore, we present a resource disaggregation scheme to improve the cluster-wide resource utilization of scientific workflows.

We implement this disaggregation scheme in **MemFSS**, building on our previous work, MemFS [5] and MemEFS [6]. Compared to a static reservation size required by an application's data footprint, MemFSS uses a smaller set of *own* nodes to run computations and to store data. In addition, MemFSS extends its storage space by scavenging for unused memory in *victim* cluster reservations of other tenants. The key mechanism enabling this change is the new hashing scheme used in MemFSS, adapted to variable storage amounts per node, whereas Mem(E)FS employed a uniform distribution.

Our contributions are:

- We present a *memory scavenging* design for an in-memory distributed file system.
- We show that the memory scavenging approach is feasible and it incurs modest slowdowns in state-of-the-art high-performance computing or big-data benchmarks used as tenant-applications.
- We show that the memory scavenging approach largely reduces the compute resource consumption of typical scientific workflow applications.

The paper is structured as follows. Section II introduces an in-depth motivation and background knowledge for our memory scavenging approach. Section III explores the design space for MemFSS, while Section IV presents the in-depth evaluation of our resource disaggregation scheme. In Section V we discuss previous research related to our work. Section VI concludes the paper.

II. Motivation and Background

In this section we present the motivation and background on our work. We show that resource under-utilization is characteristic not only to running scientific workflows on clusters, but also generally, clusters and data centers are largely under-utilized in two key resources: *memory* and *network bandwidth*. Our previous work, MemFS [5], heavily relied on memory and network to improve I/O for scientific workflows. In this work, we show that a memory disaggregation approach for running workflows on MemFSS greatly improves cluster resource utilization by taking advantage of unutilized memory and network from reservations of other tenants.

A. The Achieved Parallelism of Scientific Workflows

Scientific workflows are applications composed of many tasks linked through data dependencies. Such applications are typically described by *directed acyclic graphs* (DAGs). Scheduling application DAGs is difficult since not only system-level knowledge needs to be taken into account, but also application-specific knowledge: not all tasks are ready to be scheduled at once as they need their data dependencies to be resolved.

Tasks are generally organized in different task types (stages), and the number of tasks per stage is highly variable. There are stages composed of one to tens of tasks, while, in contrast, there are highly parallel stages composed of thousands to tens or hundreds of thousand tasks. Therefore, the level of potential parallelism when running such workloads is highly variable, limiting the scalability of the application.

Recent in-depth analyses [7], [8] on real-world scientific workflows confirm this behavior and add valuable insight into the ranges of task running times. The stages that exhibit less parallelism are generally composed of longer running tasks compared to the massively parallel stages. Such tasks generally perform data aggregation (analyzing or collecting data from previous stages) or partitioning (producing data for subsequent tasks). This is the case for Montage [9], CyberShake [10], LIGO [11], SIPHT [12], Epigenomics [13], and SoyKB [14] workflows.

In conclusion, executing real-world workflows largely limits scalability. This is not only due to the inherent shape of the application DAG, but also due to the large variability of task runtimes between various stages. Therefore, executing such applications on compute clusters severly under-utilizes the reserved resources.

B. Cluster Resource Utilization

We gathered several studies that present in-depth analyses on cluster, data center or computing framework resource utilization. Our main findings are summarised in Table I. We start by presenting the findings on memory utilization, and then end with the network utilization.

TABLE I
CPU, MEMORY AND NETWORK UTILIZATION SUMMARY OF VARIOUS SURVEYS.

Study	CPU	Memory	Network
Google Traces	60%	50%	N/A
Facebook	N/A	19% (median)	N/A
Taobao	$\leq 70\%$	$\in (20\%, 40\%)$	10-20MB/s
Mesos	$\leq 80\%$	$\leq 40\%$	N/A
Graph Processing Platforms	$\leq 10\%$	$\leq 50\%$ (mean)	$\leq$ 128Mbit/s
Commercial Cloud Datacenters	N/A	N/A	$\leq 20\%$ bisection bandwidth used

1) Memory Utilization: A trace-based analysis [3] of a large size Google cluster reports that memory utilization is approximately 50%, while CPU usage 60%. In a MapReduce Facebook cluster, memory was also often underutilized: the median and 95th percentile memory utilizations are 19% and 42%, respectively [15]. The same behaviour is noticed in another Hadoop cluster (Taobao) [16]: average memory usage is between 20%-40%, while CPU usage is up to 70%. Here, the authors notice a strong correlation between the usage patterns: a burst in CPU utilization triggers a burst in memory usage. An extensive study on graph processing platforms [17] shows that all frameworks utilize less than 50% of the memory. Mesos [4], a platform for sharing cluster resources between multiple computing frameworks, is able to increase memory utilization from 20% to 40%, leaving large amounts of memory unutilized.

2) Network Utilization: Regarding network bandwidth utilization, a study [18] that monitored multiple public and private data centers reports that at most 20% of the bisection bandwidth is utilized. In educational and private data centers at least 50% of the traffic is extra-rack, while in commercial and cloud data centers, at least 75% of the traffic is intra-rack. In such data centers, rack-local traffic is higher due to the nature of the applications: MapReduce-like frameworks are optimized for data-locality.

Other studies also stress that bandwidth utilization is low in MapReduce-like frameworks. In the Taobao Hadoop cluster [16], the bandwidth utilized is between 10-20 MB/s. In [19], a comparative study of join algorithms for Hadoop, the achieved bandwidth is at most 50 MB/s. Furthermore, an extensive performance analysis [20] study of Hadoop and Spark states that network is never a bottleneck for such data processing platforms. A similar behavior is noticed in graph processing frameworks, where bandwidth utilization is not higher than 128 Mbit/s [17].

C. MemFS

Previously, we introduced MemFS [5], an in-memory runtime distributed file system that highly improves the I/O performance of scientific workflows. MemFS stores data in main memory and is able to saturate premium network (InfiniBand, 10G Ethernet) bandwidth. Using a consistent hashing [21] scheme, MemFS spreads data evenly across cluster reservation nodes. Hence, MemFS achieves balanced storage and network traffic.

As memory-to-disk ratio is still low in current clusters, a large number of nodes needs to be deployed such that intermediate data fits in memory. Since scientific workflows often have inherently poorly scalable stages, the reserved CPU cycles will largely be wasted, meaning an even lower resource utilization.

However, as cluster memory and network are also heavily under-utilized, we designed a software memory disaggregation scheme in which we allocate a small number of nodes to our system and extend its storage space by scavenging for memory in other tenants' allocations. Thus, the total cluster resource utilization increases.

III. MemFSS Design

In this section we describe our design decisions when building a memory-scavenging distributed file system.

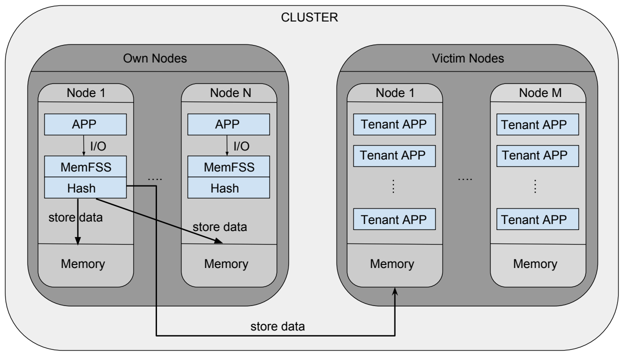


Fig. 1. Scavenging MemFSS Design.

A. The MemFSS System

Previously, in both MemFS [5] and MemEFS [6] all system nodes had a dual role: running tasks and storing data. Also, all nodes received an equal share of both the computation and the data. However, in a memory-scavenging scenario this setup is unfeasible: we want to use only a small share of memory and bandwidth from the scavenged nodes such that their tenant applications are not disrupted.

Hence, in MemFSS we impose a node hierarchy. First, the user reserves a set of *own nodes* through the cluster reservation system. Since these nodes fully belong to the user, they will both run tasks and store data. Second, MemFSS makes use of *victim* or *scavenged* nodes. These nodes are only used to augment the storage space of the *own nodes* and do not run tasks. Figure 1 depicts the overall design of MemFSS.

We propose two mechanisms for selecting the *victim* nodes, both requiring minor modifications to the cluster reservation system. First, the cluster users could, on a voluntary basis, register their nodes to a secondary queue along with the amount of memory that could be accessed by MemFSS. Second, the cluster administrator could enforce all used nodes to be registered to a secondary queue, and cap the amount of scavenged memory MemFSS may use (e.g. 10GB). Then, whenever the tenant applications would need more memory, a monitoring process would send a signal to MemFSS to free its memory and remove itself from that node.

B. Hashing

The key problem behind MemFSS' data placement is to find a hashing scheme that accommodates sets of nodes with different capacities. Intuitively, we need to control the amount of data (and, hence, of network traffic) that is distributed across the *victim nodes* such that the tenant applications are not severely affected. We achieve this by implementing a modified *Highest Random Weight* (HRW) [22] hashing protocol.

The original version of the HRW protocol works as follows. Given a set of servers $S = \{S_1, S_2, ..., S_n\}$ and a hash function $H(s, k)$, the placement of key K is decided by computing all the $H(S_i, K)$ values and choosing the server which yields the largest hash value. This hashing scheme achieves the same minimal disruption properties as consistent hashing [21]. This means that whenever a server is added or removed, if there are M keys mapped to N servers, only $O(M/N)$ keys need to be remapped. The decision mechanism yields an $O(n)$ computational complexity.

Based on the HRW protocol, we design a hashing scheme that allows us to control the amount of data that is sent to different subsets of nodes. In our case, the cluster nodes are split into two classes - *own* and *victim* nodes. Therefore, we apply the hashing scheme to classes of nodes, reducing the computational complexity of the decision mechanism. However, as the HRW algorithm achieves uniform data placement, we modify the HRW hash function such that it takes into account *class weights*. Thus, whenever a key is hashed, we compute two values $H(C_{own}, W_{own}, key)$ and $H(C_{victim}, W_{victim}, key)$. Whichever class yields the highest value will store the key. In our implementation, we keep the hash function reported in [22] from which we subtract a weight. Using weights allows us to control the amount of data that is sent to the victim nodes: larger weights for the *victim* class generate lower loads, while smaller weights yield higher loads. This scheme can be generalized to an arbitrary number of classes, allowing for multiple types of *victim* classes.

Once the class to which a key belongs to is decided, we need to choose a node from that class to store the key. This second mapping needs to maintain a similar load across all nodes in a class to ensure balanced and predictable network traffic. In this phase, we use the original HRW algorithm.

Thus, we employ a two-layer hash mechanism. The former, weight-based, decides which class of nodes will store a key. The latter distributes the key uniformly to the nodes in the class. Our multi-layer hashing overlay is similar to the mechanism presented in [23]. This is an optimization to the initial HRW algorithm, achieving $O(logN)$ decision time, albeit not considering weights or skewed data distributions.

C. POSIX (FUSE) Interface

Typical scientific workflows are composed of tasks represented by legacy binaries which perform POSIX I/O operations. To support such applications, we have implemented a lightweight FUSE [24] layer that handles I/O operations and implements the HRW hashing protocol previously discussed. Using the hash-based decisions, reading and writing data operations are then forwarded to the data store. This FUSE layer is also responsible for striping the files into smaller pieces of data such that we achieve load balance within nodes in the same class (own or victim). The HRW protocol is then applied to each file stripe to decide which node will store it. In this memory-scavenging scenario, only the *own nodes* mount the FUSE file system, since only these nodes run tasks.

D. Data & Metadata Placement

MemFSS uses Redis [25] as the in-memory data store. In our setup, both *own* and *victim* nodes run redis processes. We store both *data* and *metadata* in Redis. The former is stored in its raw format, as generated by the application. The latter contains file system organization information: directory structure, file sizes, number of file stripes and the HRW weights we used to decide the file stripe placement. Storing the weights in the metadata is needed to support dynamic additions of subsequent *victim* node classes, if necessary (e.g. to further extend the storage space).

Data is spread across the entire storage space based on the HRW hashing scheme. Metadata storage is restricted to the *own* node class and we use a simple modulo hashing scheme to distribute it. The reason for this design decision is threefold. First, to support dynamic additions of *victim* classes we need a simpler scheme to identify and store metadata while the HRW weights could change. Second, we believe the *own* nodes less likely to fail or run out of memory since we control all applications running on them. Third, metadata operations are generally latency-bound and thus, we try to keep metadata closer to the nodes that run applications. *Victim* nodes could suffer from higher latencies since they run other applications. Even though latency might be higher in the *victim* nodes, we argue this is not a problem for storing or retrieving data, as these are mostly bandwidth-bound operations. As shown in Section II, there is enough bandwidth available in the system.

E. Fault-Tolerance

The HRW hashing offers a simple, yet effective opportunity to achieve fault-tolerance. When computing the second HRW hashing stage (i.e. to decide which node from a class stores the data), we can replicate the data on the servers that yield the second (and third) highest value(s). A similar replication strategy can be applied when storing metadata.

However, since we store data in main memory, replication could be a prohibitive strategy, as memory-to-disk ratio is still low in current systems. A more suitable approach, which entails less redundancy, is represented by erasure coding [26]. We are currently working on implementing such an approach, but presenting it is outside the scope of this paper.

F. Data Privacy & Isolation

Since MemFSS runs Redis servers on nodes that run applications of other users, we need to ensure isolation and privacy for our application data. We achieve these features using two mechanisms. First, we use the *authentication* feature of Redis such that only the clients residing on the *own nodes* could send requests. Second, we run the Redis processes in the *victim* nodes inside Linux containers [27]. This offers a lightweight isolation and virtualization mechanism in which we can specify, with a fine granularity, the amount of resources (CPU, memory, network) that could be used by our Redis processes.

IV. Evaluation

In this section, we present an in-depth evaluation of our memory scavenging framework. First, we show the baseline, that is the worst-case scenario overhead that MemFSS incurs in the *victim nodes* when various amounts of data are stored in the *own nodes*. Second, we show the overhead incurred by MemFSS scavenging memory from *victim nodes* running as tenant applications real-world high-performance computing and big data benchmarks. Finally, we analyze the reduction in system resource consumption delivered by our proposed mechanism.

A. Experimental Setup

The platform on which we executed the experiments is our local DAS-5 [28] cluster. DAS-5 consists of 68 nodes, each equipped with dual 8-core Intel E5-2630v3 (two hyper-threads per core) CPUs and 64GB memory. The nodes are interconnected by two networks: an 1Gbps Ethernet, and an 54 Gbps FDR InfiniBand (approx. 3GB/s for IPoIB). For all our experiments we have used the latter. Our experiments were run on 40 compute nodes that were split as follows: 8 *own* nodes running scientific workflows on top of MemFSS, and 32 *victim* nodes running tenant applications. During runtime, the 8 MemFSS nodes were scavenging for memory in the 32 *victim* nodes.

1) MemFSS Workloads: The workloads we used for MemFSS are the following:

- (i) a bag of 2048 *dd* tasks, that each write 128MB;
- (ii) the Montage [9] workflow;
- (iii) the BLAST [29] workflow.

We consider these three workloads representative for a wide range of scientific workloads. First, the *dd* tasks are an I/O bound micro-benchmark, generating the largest load onto the *victim* nodes. This shows the upper bound of how much overhead could be induced by memory scavenging. Second, the *Montage* and *BLAST* workloads are real-world applications that show the expected behavior of the system when doing memory scavenging.

The stages of *Montage* are a combination of CPU- and I/O-bound tasks. Although *BLAST* tasks are mostly CPU-bound, they also show moderate memory and I/O utilization. In terms of runtime, *Montage* tasks are short (order of seconds), while *BLAST* tasks are longer (tens of seconds to minutes). Regarding data size, *Montage* generates small files (1-4MB). *BLAST* deals with much larger files (hundreds of MB).

2) Tenant Workloads: The workloads representing tenant applications (thus running on the *victim* nodes) are well-known real-world high-performance computing and big data benchmarks:

- (i) HPCC [30] - composed of several MPI [31] applications that assess several performance aspects of computing systems: CPU speed, memory bandwidth, network bandwidth and latency;
- (ii) HiBench [32] - composed of several data processing applications with different characteristics.

These benchmarks and their respective stages (e.g. map, shuffle, reduce) are either CPU-, network-, or I/O-bound. We run the HiBench suite on both Hadoop [33] and Spark [34] data processing frameworks.

We measure the upper bound on the performance degradation induced by MemFSS by both tuning the benchmark suites for maximum performance and using large input sizes. Both benchmark suites were configured to use all available cores (and hyperthreads). For HPCC we used an input size that generated a load of at most 48GB memory per node, such that MemFSS could scavenge at most 10GB per *victim* node. For HiBench, we used the *big data* input size for both Hadoop and Spark, while scavenging at most 10GB memory per *victim* node. In the Spark use case, we have allocated only 48GB memory per node for the Spark workers.

B. Scavenging Overhead Baseline

Figure 2 shows the baseline for running MemFSS on a set of $n = 8$ *own nodes* and $m = 32$ *victim nodes*. The *victim* nodes are only running the data store components, without any tenant application. This experiment shows the overhead incurred by MemFSS on the victim nodes. We ran 5 scenarios: $\alpha \in \{0\%, 25\%, 50\%, 75\%, 100\%\}$ of the data residing on the own nodes, while the rest was evenly spread among the victim nodes. Each experiment consists of a bag of 2048 `dd` tasks, each writing 128MB data. This sums up to 256GB.

Figures 2a - 2e show the CPU and network utilization in both *own* and *victim* nodes while the percentage of data stored in own nodes increases. As expected, we notice that both CPU and network bandwidth load decrease for the *victim* nodes when the amount of data stored in the *own* nodes increases. However, we notice that the CPU load of the victim nodes is never higher than 5%, while the network bandwidth utilized is always lower than 500MB/s (16%). This is an important result, since it shows that the memory scavenging mechanism of MemFSS only consumes a small fraction of the victim nodes' resources.

Figure 2f summarizes these 5 scenarios. We notice that the best performing scenario is that where 25% of the data reside on the *own nodes*. The intuition is that in each scenario, nodes belonging to the same subset (i.e. own, victim) hold (approximately) the same amount of data. In the particular case of the 25%, the ratio between data stored on an own node and a victim node $\frac{\alpha \times m}{(1-\alpha) \times n}$ is closest to 1 among the considered cases. This means the 25% case outperforms the other setups in terms of load balancing. As previously shown [5], load-balance for in-memory storage systems when running MTC applications is key for performance.

C. Real-World Application Slowdown

We now assess the overhead incurred by memory scavenging on real-world applications. Figure 3 shows the slowdown induced by MemFSS scavenging memory from the HPCC benchmark suite. Figure 3a shows the slowdown when MemFSS stores 25% of the data on its own nodes, while Figure 3b for the 50% case. We use only these two volumes of data because they achieve a good performance (Figure 2f) and they allow the user to achieve the largest storage space (data resides on both own and victim nodes). As HPCC reports many performance measurements, for brevity, we only plot the categories suggested on the HPCC website [1]. We measured the HPCC slowdown induced by running all the MemFSS applications: Montage, BLAST, and dd.

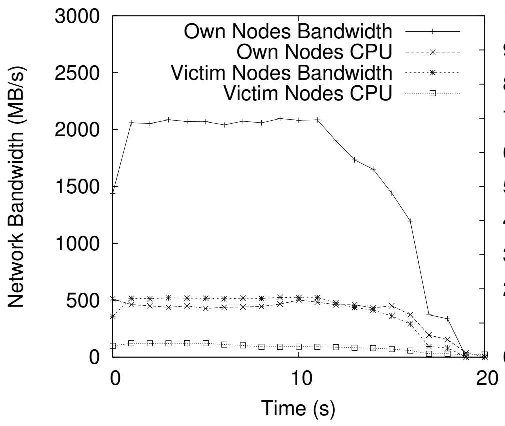
Our results show that most of the individual benchmarks are slowed down by less than 10%. In the 25% case (Figure 3a), only the STREAM (benchmark measuring memory bandwidth) and the latency benchmarks are slowed down by 11-12%. It is expected that these two benchmarks are slowed down more than the others since we compete with HPCC for memory (and memory bandwidth) and network. However, in the 50% case (Figure 3b), most of the numbers are lower than in the 25% case. This is expected behavior since sending less data to the victim nodes incurs less overhead. The only outlier in the 50% case is the FFT benchmark when running dd (slowdown of 13%). A more in-depth analysis for this behavior is outside the scope of this paper.

It is interesting to notice that for most benchmarks, Montage gives the smallest slowdown. This is expected since Montage has long-running sequential (data aggregation/partitioning) stages that do not read or write much data. Comparing BLAST and dd, one would expect that dd incurs the highest slowdown since it is an I/O intensive application. However, this is not the case for scavenging memory from HPCC. This behavior is explained by the fact that BLAST makes many short I/O requests, as opposed to dd, which makes larger I/O requests. These many small requests induce higher latencies into the nodes running HPCC. As the HPCC benchmarks are MPI high-performance computing applications, it is likely that they are latency-sensitive.

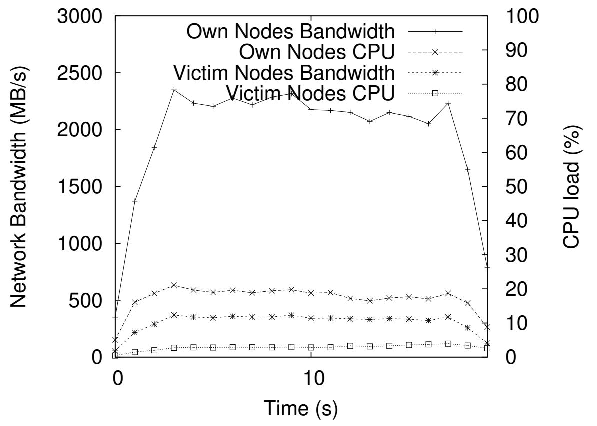
Figure 4 shows the slowdown induced by scavenging memory from the *victim* nodes while they run the HiBench suite on top of Hadoop. For brevity, we have selected from the HiBench suite 6 representative benchmarks: KMeans is mostly CPU-intensive, but also has a high I/O utilization; PageRank is CPU-bound, but has a highly variable CPU utilization; WordCount is CPU-bound, but also has a high memory usage; TeraSort is CPU-intensive in the map-phase, it utilizes a large amount of memory and it also exhibits a large network traffic in the shuffle phase; The DFSIO-read and DFSIO-write benchmarks are I/O intensive, but also generate a large amount of network traffic.

Figures 4a and 4b show the slowdown induced by memory scavenging when MemFSS stores 25% of the data on the *own*

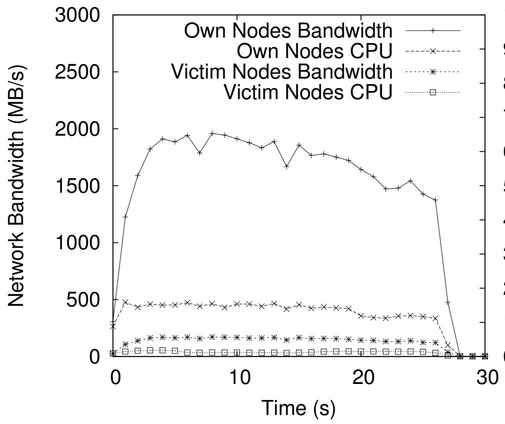
[1] http://icl.cs.utk.edu/hpcc/hpcc_results_kiviat.cgi



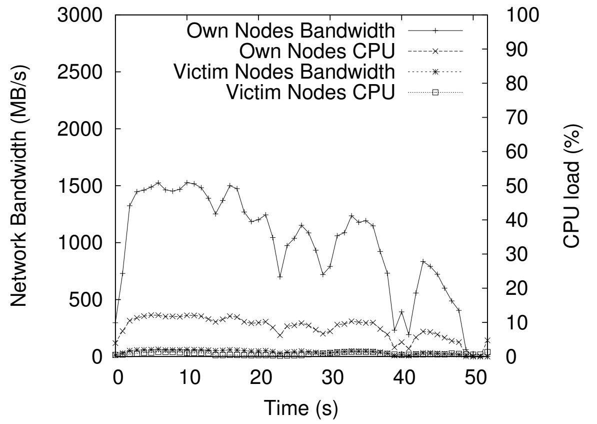
(a) 0% data own nodes.



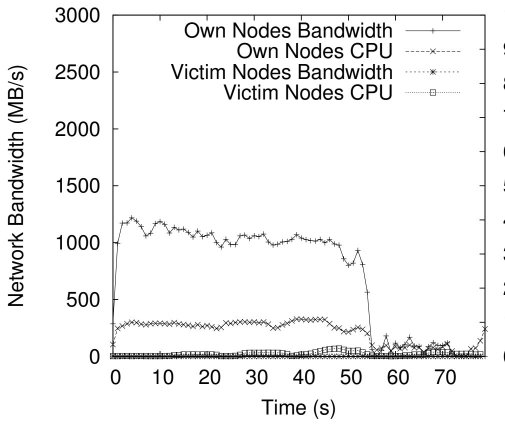
(b) 25% data own nodes.



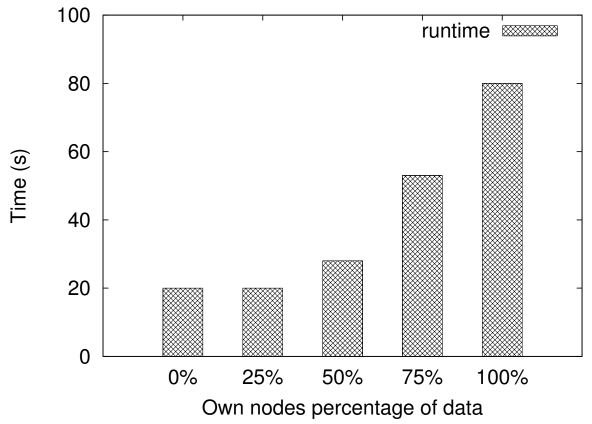
(c) 50% data own nodes.



(d) 75% data own nodes.



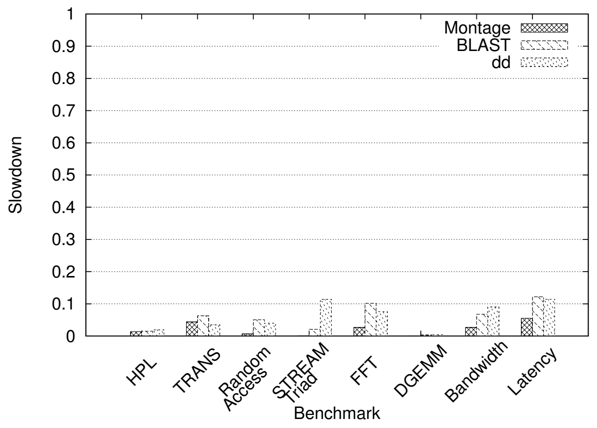
(e) 100% data own nodes.



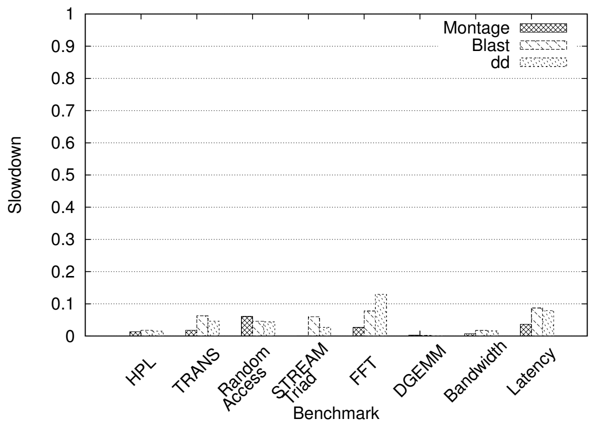
(f) Runtimes.

Fig. 2. Average CPU, bandwidth utilization and runtime comparison when different amounts of data are kept in *own nodes* and the rest in *victim nodes*. We have used 8 *own nodes* and 32 *victim nodes*.

nodes, and, respectively 50% of data. As expected, for most benchmarks, the overhead incurred by memory scavenging is lower than 10%. However, in the 25% case, for the TeraSort benchmark, both BLAST and dd induce a higher overhead: 16% and 26%, respectively. This is because TeraSort utilizes a large amount of memory and generates a very large network traffic during the shuffle phase. Thus, MemFSS' scavenging competes with TeraSort for both memory and network. In the 50% case, the slowdown is much smaller: 8% for BLAST and 15% for dd. Another benchmark that is slowed down more than 10% is the DFSIO-read benchmark. When Hadoop reads data from HDFS [35], the data is cached on the node's page

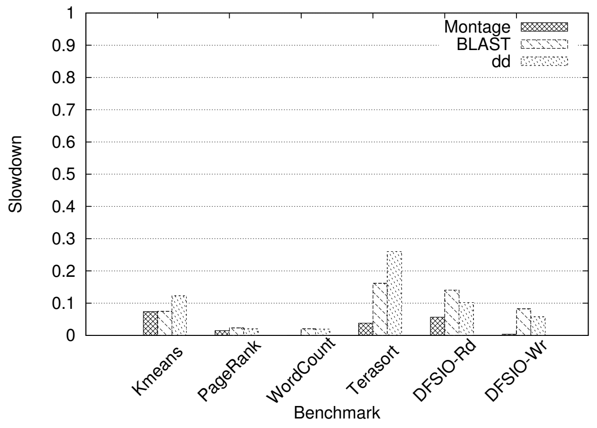


(a) MemFSS own nodes store 25% of the data.

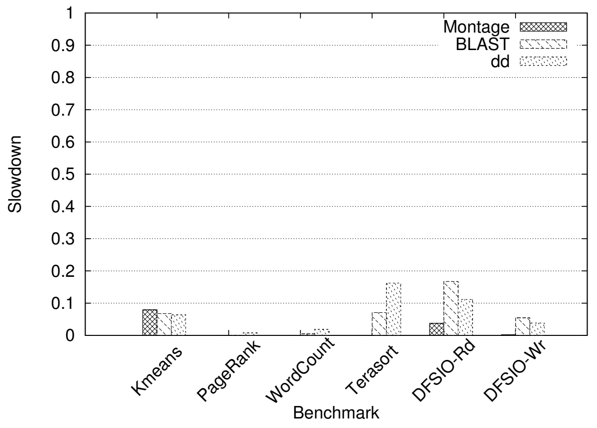


(b) MemFSS own nodes store 50% of the data.

Fig. 3. HPCC slowdown induced by memory scavenging.



(a) MemFSS own nodes store 25% of the data.



(b) MemFSS own nodes store 50% of the data.

Fig. 4. HiBench Hadoop slowdown induced by memory scavenging.

cache. Therefore, we are competing with the page cache for memory. This behavior induces a slowdown for the DFSIO-Read benchmark. It is also important to notice that for most benchmarks, the slowdown is lower in the 50% case because storing less data in the *victim* nodes induces less overhead.

Figure 5 shows the slowdown induced by MemFSS' scavenging mechanism when Spark runs the HiBench suite (from which we excluded the DFSIO benchmarks, as they are not yet implemented for Spark). Because Spark is also an in-memory data processing platform, for this experiment, we stored only 50% of the data in the *victim* nodes. Storing more data into the *victim* nodes is not feasible as it would decrease the memory space available for Spark.

We notice that for most of the benchmarks, the slowdown is much larger than for Hadoop HiBench or HPCC. This is expected, since Spark is itself relying on memory to improve performance. As such, we compete with Spark not only for network, but also for memory in both capacity and bandwidth. Because of this, we also slow down the JVM garbage collector.

To give an overview of our experiments, in Figure 6, we show the average slowdown given by memory scavenging for all the workloads. The results show that for both HPCC and

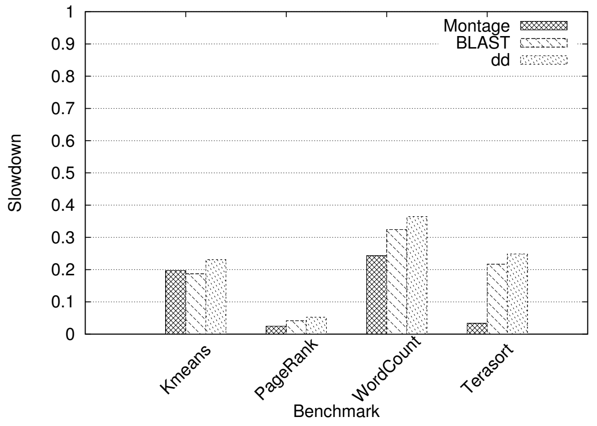


Fig. 5. HiBench Spark - slowdown induced by memory scavenging when MemFSS own nodes store 50% of the data.

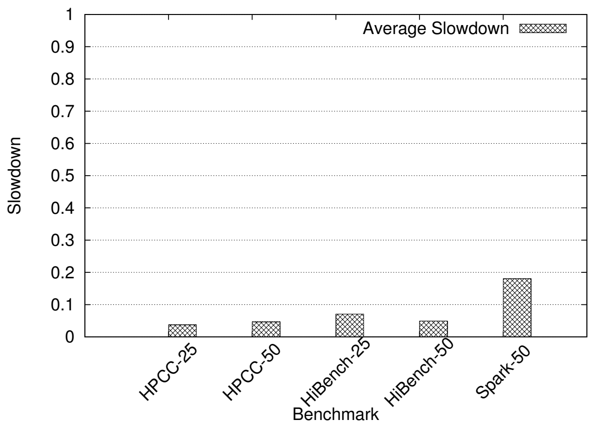


Fig. 6. Average Slowdown induced by memory scavenging.

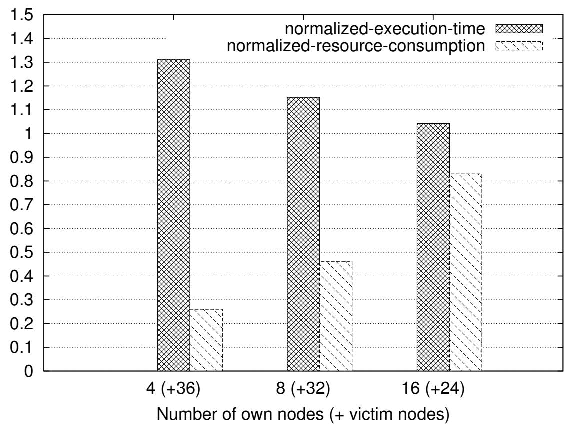


Fig. 7. Normalized execution time and normalized resource utilization when doing memory scavenging in various own-nodes - victim-nodes setups.

Hadoop HiBench, in both the 25% and 50% cases, the average slowdown is below 10%. The only outlier is the Spark case, which employs an average slowdown of 18%.

D. Resource Consumption Reduction

In the previous sections we have shown the implications of memory scavenging for the *victim* applications. In contrast, here we analyze how memory scavenging reduces the resource consumption for the scientific workflows tehmselves. Table II shows a typical scenario: to run a large instance of Montage, a user needs to deploy 20 nodes such that all data fits into memory (total data size is approximately 1TB). This leads to a running time of 4521 seconds. When doing memory scavenging, Montage can run on MemFSS which uses $n \in \{4, 8, 16\}$ *own* nodes, while increasing the storage space by scavenging memory from $m = 40 - n$ *victim* nodes. In our test cases, the total running time is only 4%-31% higher due to the limited parallelism of Montage. However, its resource consumption is reduced by 17%-74%. Figure 7 summarizes these results. Here, we plot the normalized runtime and resource consumption, compared to the static run with 20 nodes. As expected, the more *own nodes* we use, the lesser resources we save. Moreover, as the number of *own nodes* increases, the runtime is closer to the static run, as there are more worker nodes to run application tasks.

TABLE II
RESOURCE UTILIZATION IMPROVEMENT

Application	No. of Nodes	Runtime (s)	Node-hours
Montage (standalone)	20	4521	25.11
Montage (standalone)	< 20	Unable to run, data does not fit	N/A
Montage (scavenging)	4 (+ scavenging)	5932	6.59
Montage (scavenging)	8 (+ scavenging)	5213	11.58
Montage (scavenging)	16 (+ scavenging)	4711	20.93

E. Summary

When evaluating MemFSS we have used microbenchmarks to assess the resource load induced in the victim nodes by our scavenging approach. The results show that the load is proportional with the amount of data sent to the victim nodes. The CPU load of the scavenged nodes is never higher than 5%, while the network bandwidth load is never higher than 16%. These results are confirmed when on the victim nodes we run real-world benchmark suites (HPCC, HiBench). For both cases, the average slowdown for the victim application is less than 10%. The only case when the induced slowdown is higher (18%) is when the scavenged application is Spark. However, this is an expected behavior since Spark is an in-memory data processing framework. These results show that memory scavenging is feasible even when the victim applications are well established, highly tuned for performance benchmark suites. Furthermore, when considering scientific workflows, MemFSS largely reduces compute resource consumption. This is an important result as the reduced resource consumption leads to overall energy savings and shorter queueing times in clusters.

V. RELATED WORK

A. Cycle Scavenging - Volunteer Computing

Scavenging for resources has been traditionally driven by the CPU cycles requirements of large-scale applications. As idle CPU cycles are used for large-scale computations, several design choices have to be considered: access policies, resource scheduling, job priority, participation incentives. Since Condor [36], many cycle scavenging platforms, such as BOINC [37], SETI @ home [38], Entropia [39] or Koala [40], have been implemented to cater for various design requirements. However, they considered the CPU cycles and memory required by a scavenging job as a single scavenged resource, since the contemporary network performance would have prohibited a finer separation.

B. Remote Memory Usage

Using remote memory swapping to prevent local swapping to disk is a strategy explored in several studies [41], [42]. In contrast to MemFSS, these solutions assume cluster-wide reservations and focus only on harnessing the entire memory capacity of such reservations. In contrast, MemFSS focuses on optimizing the utilization of all available resources in a cluster.

Muhleisen et al. [43] introduced a solution leveraging the remote memory and high-speed interconnects of a cluster to improve the performance of main-memory databases. This solution is similar to MemFSS in that it stores intermediary data (the databases' working directory) in a file system built using the remote memory of the nodes registered as *Providers*. It also does not require any changes to the OS, nor does it need special hardware. However, MemFSS differs significantly in that it targets both improving application performance *and* optimizing system-wide resource utilization.

In terms of cluster resource disaggregation, Hou et al. introduced the design of a hardware-level solution [44]. As a software-level solution, MemFSS adapts to various environments without incurring significant overhead.

C. Weighted Hashing

The idea of using two hash functions to accommodate servers with different capacities is common to several projects [45], [46], [47]. The first hash function maps the nodes to a continuous interval $[0, 1)$, while the second determines the location of the keys by mapping them in the same interval. Brinkmann et al. introduce two adaptive hashing strategies [45] to redistribute keys among nodes when either the capacities of the nodes, the number of nodes or the number of keys change. Each node manages multiple virtual bins, and each virtual bin handles one sub-interval with a length proportional to its capacity and a stretch factor. Schindelhauer et al. improves the load balancing in heterogeneous DHTs by choosing nodes for keys based on weights [46]. Each node is assigned a positive weight and keys are distributed to it with a probability proportional to the node's weight and inversely proportional to the sum of all node weights. Previously, in MemEFS [6], we adapted Y0's algorithm [47] to provide elasticity and cope with node heterogeneity.

For MemFSS we choose a simpler, yet effective weighted hashing scheme: the HRW [22] protocol. The motivation for choosing this algorithm is twofold. First, in the aforementioned consistent hashing schemes, when new nodes are added, data must be moved during runtime; otherwise, the hash would give incorrect locations when performing look-ups. With HRW, data does not need to be moved. If a piece of data is not found, the data could still be found by checking the following servers in the hash list. In this way, we can employ a *lazy* movement of data: once a key is found on a server that yields a lower hash value, it can be moved to the right server. In this way, there is no need to stop the computation such that the data placement is correct. Second, for practical reasons, in our system virtual bins would be implemented as redis processes. Using multiple redis processes per node means adding significant memory and CPU overhead. HRW allows us to use only one redis process per node, minimizing the data store overhead.

VI. Conclusions

Compute clusters are the de-facto platform for running scientific or business-oriented workloads, each with their own characteristic resource requirements. Therefore, clusters are built following a one-size-fits-all design, leading to under-utilized components. Many studies point out that, generally, in clusters, there are two types of resources that are under-utilized - *memory* and *network (bandwidth)*. In our previous work, we have developed an in-memory runtime distributed file system for scientific workflows. In such a system, the key resources for performance are memory and network. Complementary to typical cluster workloads, when running scientific workflows, CPUs are under-utilized due to the intrinsic application structure, which limits scalability: large parallel stages are followed by (long-running) data aggregation or partitioning stages.

In this paper, we presented the design and implementation of MemFSS, an in-memory distributed file system which extends its storage space with cluster resources disaggregated through memory scavenging. MemFSS uses a smaller set of *own* nodes than that required by an application's data footprint. Our evaluation, using real-world scientific workflows, and state-of-the-art high-performance computing and big data benchmarks as tenant applications, shows that memory disaggregation is both feasible and beneficial. First, most tenant applications exhibit slowdowns below 10%. Even in the worst-case scenario, i.e. the memory-hungry Spark processing framework, the slowdown is below 20%. Second, MemFSS improves compute resource consumption by wide margins.

Encouraged by these results, we envision a larger area of applicability for (memory) disaggregation techniques. In both public and private infrastructure, the large amount of unused memory could be collected in datacenter-wide pools of fast storage. In clouds, disaggregated memory could be presented to users in the form of network-attached fast storage, or could be used as cache for virtual machine management operations. As future work, we will continue exploring the applicability of memory disaggregation to improve the performance and resource utilization at both application- and system-level.

Acknowledgments

This work is partially funded by the Dutch public-private research community COMMIT/. The authors would like to thank Kees Verstoep for providing excellent support on the DAS-5 clusters.

References

[1] S.-H. Chiang and M. K. Vernon, "Characteristics of a large shared memory production workload," *Job Scheduling Strategies for Parallel Processing*. Springer, 2001, pp. 159–187.

[2] H. Li, D. Groep, and L. Wolters, "Workload characteristics of a multi-cluster supercomputer," *Job Scheduling Strategies for Parallel Processing*. Springer, 2004, pp. 176–193.

[3] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch, "Towards understanding heterogeneous clouds at scale: Google trace analysis," *Intel Science and Technology Center for Cloud Computing, Tech. Rep*, p. 84, 2012.

[4] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center." *NSDI*, Vol. 11, 2011, pp. 22–22.

[5] A. Uta, A. Sandu, and T. Kielmann, "Overcoming data locality: An in-memory runtime file system with symmetrical data distribution," *Future Generation Computer Systems*, Vol. 54, pp. 144–158, 2015.

[6] A. Uta, A. Sandu, S. Costache, and T. Kielmann, "MemEFS: an Elastic In-Memory Runtime File System for eScience Applications," *e-Science (e-Science), 2015 IEEE 11th International Conference on*. IEEE, 2015, pp. 465–474.

[7] G. Juve, A. Chervenak, E. Deelman, S. Bharathi, G. Mehta, and K. Vahi, "Characterizing and profiling scientific workflows," *Future Generation Computer Systems*, Vol. 29, no. 3, pp. 682–692, 2013.

[8] R. F. da Silva, G. Juve, M. Rynge, E. Deelman, and M. Livny, "Online task resource consumption prediction for scientific workflows," *Parallel Processing Letters*, Vol. 25, no. 03, p. 1541003, 2015.

[9] J. C. Jacob, D. S. Katz, G. B. Berriman, J. C. Good, A. Laity, E. Deelman, C. Kesselman, G. Singh, M.-H. Su, T. Prince *et al.*, "Montage: a grid portal and software toolkit for science-grade astronomical image mosaicking," *International Journal of Computational Science and Engineering*, Vol. 4, no. 2, pp. 73–87, 2009.

[10] "SCEC project, Southern California Earthquake Center," http://www.scec.org/, 2015.

[11] D. A. Brown, P. R. Brady, A. Dietz, J. Cao, B. Johnson, and J. McNabb, "A case study on the use of workflow technologies for scientific analysis: Gravitational wave data analysis," *Workflows for e-Science*. Springer, 2007, pp. 39–59.

[12] J. Livny, H. Teonadi, M. Livny, and M. K. Waldor, "High-throughput, kingdom-wide prediction and annotation of bacterial non-coding rnas," *PloS one*, Vol. 3, no. 9, p. e3197, 2008.

[13] "epigenomics," http://epigenome.usc.edu, 2016.

[14] T. Joshi, B. Valliyodan, S. M. Khan, Y. Liu, J. M. dos Santos, Y. Jiao, D. Xu, H. T. Nguyen, N. Hopkins, M. Rynge *et al.*, "Next generation resequencing of soybean germplasm for trait discovery on xsede using pegasus workflows and iplant infrastructure," 2014.

[15] G. Ananthanarayanan, A. Ghodsi, A. Wang, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica, "Pacman: coordinated memory caching for parallel jobs," *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 2012, pp. 20–20.

[16] Z. Ren, X. Xu, J. Wan, W. Shi, and M. Zhou, "Workload characterization on a production hadoop cluster: A case study on taobao," *Workload Characterization (IISWC), 2012 IEEE International Symposium on*. IEEE, 2012, pp. 3–13.

[17] Y. Guo, M. Biczak, A. L. Varbanescu, A. Iosup, C. Martella, and T. L. Willke, "How well do graph-processing platforms perform? An Empirical Performance Evaluation and Analysis (technical report)," http://www.ds.ewi.tudelft.nl/fileadmin/pds/reports/2013/PDS-2013-004-4.pdf.

[18] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*. ACM, 2010, pp. 267–280.

[19] S. Blanas, J. M. Patel, V. Ercegovac, J. Rao, E. J. Shekita, and Y. Tian, "A comparison of join algorithms for log processing in mapreduce," *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 2010, pp. 975–986.

[20] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B.-G. Chun, "Making sense of performance in data analytics frameworks," *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, 2015, pp. 293–307.

[21] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin, "Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web," *Twenty-ninth annual ACM symposium on Theory of computing*. ACM, 1997, pp. 654–663.

[22] D. G. Thaler and C. V. Ravishankar, "Using name-based mappings to increase hit rates," *IEEE/ACM Transactions on Networking (TON)*, Vol. 6, no. 1, pp. 1–14, 1998.

[23] W. Wang and C. V. Ravishankar, "Hash-based virtual hierarchies for scalable location service in mobile ad-hoc networks," *Mobile Networks and Applications*, Vol. 14, no. 5, pp. 625–637, 2009.

[24] M. Szeredi *et al.*, "FUSE: Filesystem in userspace," *http://fuse.sourceforge.net/*, 2016.

[25] "Redis," http://redis.io, 2016.

[26] A. G. Dimakis, P. Godfrey, Y. Wu, M. J. Wainwright, and K. Ramchandran, "Network coding for distributed storage systems," *Information Theory, IEEE Transactions on*, Vol. 56, no. 9, pp. 4539–4551, 2010.

[27] "Lxc," https://linuxcontainers.org, 2016.

[28] "DAS-5, The Distributed ASCI Supercomputer," http://www.cs.vu.nl/das5/, 2016.

[29] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, "Basic local alignment search tool," *Journal of molecular biology*, Vol. 215, no. 3, pp. 403–410, 1990.

[30] P. R. Luszczek, D. H. Bailey, J. J. Dongarra, J. Kepner, R. F. Lucas, R. Rabenseifner, and D. Takahashi, "The HPC Challenge (HPCC) benchmark suite," *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, 2006, p. 213.

[31] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: portable parallel programming with the message-passing interface*. MIT press, 1999, Vol. 1.

[32] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang, "The hibench benchmark suite: Characterization of the mapreduce-based data analysis," *New Frontiers in Information and Software as Services*, 2011, pp. 209–228.

[33] T. White, *Hadoop: The definitive guide*. " O'Reilly Media, Inc.", 2012.

[34] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: cluster computing with working sets," *USENIX conference on Hot topics in cloud computing*, 2010, pp. 10–10.

[35] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop Distributed File System," *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*. IEEE, 2010, pp. 1–10.

[36] M. J. Litzkow, M. Livny, and M. W. Mutka, "Condor-a hunter of idle workstations," *Distributed Computing Systems, 1988., 8th International Conference on*. IEEE, 1988, pp. 104–111.

[37] D. P. Anderson, "Boinc: A system for public-resource computing and storage," *Grid Computing, 2004. Proceedings. Fifth IEEE/ACM International Workshop on*. IEEE, 2004, pp. 4–10.

[38] E. Korpela, D. Werthimer, D. Anderson, J. Cobb, and M. Lebofsky, "Seti@ home – massively distributed computing for seti," *Computing in science & engineering*, Vol. 3, no. 1, pp. 78–83, 2001.

[39] A. Chien, B. Calder, S. Elbert, and K. Bhatia, "Entropia: architecture and performance of an enterprise desktop grid system," *Journal of Parallel and Distributed Computing*, Vol. 63, no. 5, pp. 597 – 610, 2003, special Issue on Computational Grids. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731503000066

[40] O. Sonmez, B. Grundeken, H. Mohamed, A. Iosup, and D. Epema, "Scheduling strategies for cycle scavenging in multicluster grid systems," *Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*. IEEE Computer Society, 2009, pp. 12–19.

[41] T. Newhall, D. Amato, and A. Pshenichkin, "Reliable adaptable network ram," *Cluster Computing, 2008 IEEE International Conference on*. IEEE, 2008, pp. 2–12.

[42] H. Midorikawa, K. Saito, M. Sato, and T. Boku, "Using a cluster as a memory resource: A fast and large virtual memory on mpi," *Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on*. IEEE, 2009, pp. 1–10.

[43] H. Mühleisen, R. Gonçalves, and M. Kersten, "Peak performance: Remote memory revisited," *Proceedings of the Ninth International Workshop on Data Management on New Hardware*. ACM, 2013, p. 9.

[44] R. Hou, T. Jiang, L. Zhang, P. Qi, J. Dong, H. Wang, X. Gu, and S. Zhang, "Cost effective data center servers," *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*. IEEE, 2013, pp. 179–187.

[45] A. Brinkmann, K. Salzwedel, and C. Scheideler, "Compact, Adaptive Placement Schemes for Non-uniform Requirements," *14th Annual ACM Symposium on Parallel Algorithms and Architectures*, ser. SPAA '02. New York, NY, USA: ACM, 2002, pp. 53–62.

[46] C. Schindelhauer and G. Schomaker, "Weighted Distributed Hash Tables," *ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '05. New York, NY, USA: ACM, 2005, pp. 218–227.

[47] P. B. Godfrey and I. Stoica, "Heterogeneity and load balance in distributed hash tables," *INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies.*, Vol. 1. IEEE, 2005, pp. 596–606.