Enhancing In-memory Efficiency for MapReduce-based Data Processing

Jorge Veiga*, Roberto R. Expósito, Guillermo L. Taboada, Juan Touriño

Computer Architecture Group, Universidade da Coruña, Campus de A Coruña, 15071 A Coruña, Spain

Abstract

As the memory capacity of computational systems increases, the in-memory data management of Big Data processing frameworks becomes more crucial for performance. This paper analyzes and improves the memory efficiency of Flame-MR, a framework that accelerates Hadoop applications, providing valuable insight into the impact of memory management on performance. By optimizing memory allocation, the garbage collection overheads and execution times have been reduced by up to 85% and 44%, respectively, on a multi-core cluster. Moreover, different data buffer implementations are evaluated, showing that off-heap buffers achieve better results overall. Memory resources are also leveraged by caching intermediate results, improving iterative applications by up to 26%. The memory-enhanced version of Flame-MR has been compared with Hadoop and Spark on the Amazon EC2 cloud platform. The experimental results have shown significant performance benefits reducing Hadoop execution times by up to 65%, while providing very competitive results compared to Spark.

Keywords: Big Data, MapReduce, In-memory Computing, Garbage Collector (GC), Performance Evaluation

1. Introduction

Nowadays, the available memory of computing systems is steadily increasing, allowing applications to use very large memory spaces. However, the leveraging of these abundant memory resources is still a challenge for current programming languages, especially those based on the Java Virtual Machine (JVM), which provides an object-oriented model and automatic memory management relying on the Garbage Collector (GC). Furthermore, popular Big Data processing frameworks like Hadoop [1], Spark [2] and Flink [3] are usually implemented using JVM-based languages (e.g., Java, Scala), making an extensive use of the memory space for data processing. The use of long-lived JVM processes with large heap spaces can significantly increase the GC overhead needed for tracking and removing an expected vast number of Java objects. This issue is a serious threat for scalable data processing, hindering overall application performance [4].

Therefore, there is a need for further studying the memory efficiency of current Big Data frameworks, developing and assessing new memory management techniques in order to use the available space in a more efficient way. This

*Corresponding author: Tel: +34 881 011 212, Fax: +34 981 167 160

Email addresses: jorge.veiga@udc.es (Jorge Veiga), rreye@udc.es (Roberto R. Expósito), taboada@udc.es (Guillermo L. Taboada), juan@udc.es (Juan Touriño)

paper addresses the memory efficiency of Flame-MR [5], a Java-based MapReduce framework that improves Hadoop performance without modifying its programming APIs. Several memory optimization techniques are introduced, analyzing their impact on the GC behavior and on the performance of representative workloads. Moreover, the suitability of different implementation alternatives of the memory data buffers is evaluated. The in-memory computing capabilities of Flame-MR are also improved to avoid the writing of intermediate results to disk. Finally, the performance of the resulting implementation is compared with Hadoop and Spark on the Amazon EC2 cloud [6] using several benchmarks and memory configurations.

The rest of this paper is organized as follows: Section 2 introduces the related work and some background information about Flame-MR and MapReduce applications. Section 3 analyzes the memory efficiency of Flame-MR, presenting some optimizations and providing experimental results on a high-performance cluster. Section 4 compares the performance of these optimizations with Hadoop and Spark on a public cloud platform. Finally, Section 5 extracts the main conclusions of the paper and proposes future work.

2. Background and related work

Apache Hadoop [1] is an open-source MapReduce framework written entirely in Java, frequently used for storing and processing large datasets. It mainly consists of two components: the Hadoop Distributed File System (HDFS) [7] that stores large amounts of data among several computing nodes, and the MapReduce engine that is used to process these datasets. MapReduce is a computing model, originally proposed by Google [8], used to process an input dataset by executing two main phases, Map and Reduce. In the Map phase, the input dataset is read from HDFS and processed by the user-defined map function, which extracts the significant information from each input record. This information is generated in form of key-value pairs that are partitioned, sorted and sent to their corresponding reducers through the network. In the Reduce phase, the key-value pairs are merged to form the reduce input, which is read by the user-defined reduce function to generate the final output and store it in HDFS. Apart from MapReduce, other application types can be deployed in a Hadoop cluster by using Yet Another Resource Negotiator (YARN) [9], which was introduced in Hadoop 2 to manage the computational resources of the nodes.

Our previous work [5] presented Flame-MR, an event-driven MapReduce architecture built on top of YARN and HDFS that improves the performance of Hadoop applications without modifying their source code (i.e., it is API-compatible with Hadoop). In each MapReduce job, Flame-MR deploys a configurable number of Workers on the computing nodes, which are Java processes that execute the workload by breaking it down in a set of MapReduce operations. These operations execute the classic MapReduce phases: input, map, sort, shuffle, merge, reduce and output. The Worker uses a ThreadPool to execute them in an optimized schedule. Moreover, a DataPool allocates memory resources to the operations by creating DataBuffers. Once they are filled, these DataBuffers are stored in DataStructures in order to be read by successive operations.

Memory efficiency has been the subject of study of many previous works. Some of them [10, 11] enhanced

the scalability of the Message Passing Interface (MPI) in terms of memory usage. Although these works are not focused on Big Data processing, some of the challenges they face are similar to our case (e.g., keeping memory scalability as the number of processes grows). Regarding Big Data frameworks, there exist some previous works that have investigated their memory efficiency. In [12], the memory usage of Hadoop and Hive [13] is characterized by analyzing the memory footprint, memory bandwidth and cache misses of different workloads. The authors of [14] studied the use of large memory pages when executing Big Data applications in non-uniform memory access systems, concluding that large pages do not show significant performance improvements when the dataset is sufficiently large. The scalability of Spark has been evaluated in [15], studying whether it is better to increase the number of nodes in the cluster or improve the hardware characteristics (CPU, memory and disk) of the nodes. Although each configuration performed differently depending on the workload type, the one with faster nodes showed a better performance per watt ratio.

The development of new techniques to improve memory management in Big Data frameworks has been addressed in different ways. In [16], the authors presented a novel two-level storage system with an upper-level in-memory file system that leverages high I/O throughput and data locality, while a lower-level in-disk parallel file system provides consistency and larger capacity. Moreover, works like [17] use new storage technologies, such as Solid State Drive (SSD) disks, to alleviate bandwidth and memory requirements. Other proposals target the memory management performed by object-oriented languages (e.g., Scala, Java) in order to adapt it to the characteristics of Big Data workloads. That is the case of Yak [18], which proposes a hybrid GC approach that distinguishes between objects belonging to the control space and the data space. Yak adapts the memory management of each object type to its lifespan characteristics, being able to alleviate the overhead of GC operations in Big Data frameworks like Hadoop. Another work, Deca [19], analyzes the data objects of Spark applications to estimate their expected lifespan, allocating and releasing memory accordingly to minimize GC overheads.

Although it would be interesting to compare these alternatives with Flame-MR, these projects are not publicly available. Note that even if Yak were available, both Hadoop and Flame-MR execute the same Java code to perform the user-defined map and reduce functions, so there are no differences in the number and type of data objects that are created. However, Hadoop and Flame-MR differ in the characteristics of the JVM processes that perform data operations. On the one hand, Hadoop executes each map or reduce operation in a single-threaded process with small heap size and a short lifespan. This reduces the impact of the GC optimizations on the performance of the applications, as JVMs stop their execution before accumulating a lot of unreferenced objects. On the other hand, Flame-MR executes all the operations belonging to a Worker in the same multi-threaded process, which must thus have a larger heap size and a longer lifespan. This puts a lot of strain onto the GC, as unreferenced objects are accumulated during the entire workload. Therefore, the GC has a greater impact on performance than in the case of Hadoop, becoming a bottleneck for some workloads. The in-memory techniques proposed in this paper minimize this bottleneck by reducing the amount of unused control objects that need to be collected. As the Java memory manager continues to operate independently, the integration of state-of-the-art GC algorithms like Yak is still guaranteed. Regarding Deca,

it is a Spark-specific solution that cannot be used to improve Hadoop or Flame-MR.

The most related paper to our work is [20], in which the memory management efficiency of NativeTask [21], a Big Data framework, was enhanced by achieving a better cache hierarchy utilization and memory access parallelism. However, the optimizations presented in that paper are highly dependent on the underlying system, while our work addresses the memory efficiency problem from a higher level perspective, taking into account the portability of the techniques.

Regarding iterative workloads, the leveraging of memory resources for caching intermediate results has been addressed by works like M3R [22], which implements a key-value data store to keep data in memory. However, M3R does not support datasets larger than the available memory in the node. iMapReduce [23] not only avoids to write intermediate results to HDFS, but also minimizes task scheduling and synchronization overheads. Nevertheless, the applicability of these optimizations is limited to applications with single-job iterations.

Finally, another modification of Hadoop, called SHadoop [24], also improves the job and task execution mechanisms, although the improvement is mostly significant in jobs with short running times.

3. Memory management optimizations

The design of Flame-MR is oriented to the execution of MapReduce applications in an efficient way [5]. In order to do so, memory usage is a crucial factor for taking advantage of large memory spaces. However, the memory profiling of Flame-MR has revealed some performance-limiting factors, which are discussed in this section. First, Section 3.1 analyzes the performance overhead caused by long GC pauses, explaining several optimizations that have been developed to reduce them. Second, Section 3.2 analyzes different implementations of the in-memory buffers that contain the data to be processed. Finally, Section 3.3 focuses on iterative workloads, explaining how to avoid the writing of intermediate results to HDFS and the benefits of doing so.

The optimizations presented in each of these sections are evaluated by means of several performance experiments carried out in a multi-core cluster, Pluton. Table 1 contains the main hardware and software characteristics of Pluton. According to these characteristics, Flame-MR has been configured as shown in Table 2, which also contains the relevant configuration parameters for HDFS. Moreover, Flame-MR has been configured to use the IP over InfiniBand interface, which allows to employ this network through the IP protocol and obtain lower latencies and higher bandwidths than with Gigabit Ethernet. The version of Hadoop deployed to make use of YARN and HDFS components is 2.7.2. The experiments have been carried out using 9 nodes, corresponding with 1 master and 8 slaves.

The benchmarks that have been executed are Sort and PageRank, which are well-known MapReduce workloads. Sort is an I/O-bound workload that has been used to order a 100 GB input text dataset generated by RandomTextWriter. Both the source code of the workload and the input generator are provided by the Hadoop distribution. PageRank is an iterative application which ranks websites by counting the number and quality of the links to each one. In the experiments, the input dataset has a size of approximately 20 GB (30 Mpages), with a maximum of 5 iterations per

Table 1: Pluton node characteristics

Hardware configuration	
CPU	2 × Intel Xeon E5-2660 Sandy Bridge-EP
CPU Speed/Turbo	2.20 GHz/3 GHz
#Cores	16
Memory	64 GB DDR3 1600 MHz
Disk	1 TB HDD
Networks	InfiniBand FDR & Gigabit Ethernet
Software configuration	
OS version	CentOS release 6.4
Kernel	2.6.32-573.7.1.el6.x86_64
Java	Oracle JDK 1.8.0_45

Table 2: Configuration of Flame-MR

Flame-MR	
Workers per node	4
ThreadPool size	4
Worker heap size	10.5 GB
DataPool size	6.3 GB
DataBuffer size	1 MB
HDFS block size	128 MB
Replication factor	3

execution. The PageRank source code has been obtained from Pegasus 2.0 [25], a graph mining system built on top of Hadoop, while the input generator is DataGen, included in the HiBench benchmark suite [26].

The execution of the experiments has been automated by using the Big Data Evaluator tool (BDEv, available at `http://bdev.des.udc.es`), which is an evolution of the MapReduce Evaluator presented in [27]. The results provided in this section take into account a minimum of 10 executions for each experiment making sure to clear the OS buffer cache between each execution. Finally, the GC used is Parallel GC, which is enabled by default and provides the best throughput as the number of cores increases [28]. Although there exist several GC implementations that can provide different performance values, comparing them is out of the scope of this paper.

3.1. Garbage collection reduction

The Java Virtual Machine (JVM) provides automatic memory management by allocating available memory on the heap to objects when they are created. The programmer cannot control when this memory is released, as Java objects cannot be explicitly destroyed. Instead, the JVM tracks the objects on the heap and their references from the Java code, removing an object and releasing its memory when it has no other objects referencing to it. This process is transparently performed by the GC included in the JVM [29]. Using large heap sizes, garbage collection can consume a considerable amount of computational resources, incurring significant performance penalties and even

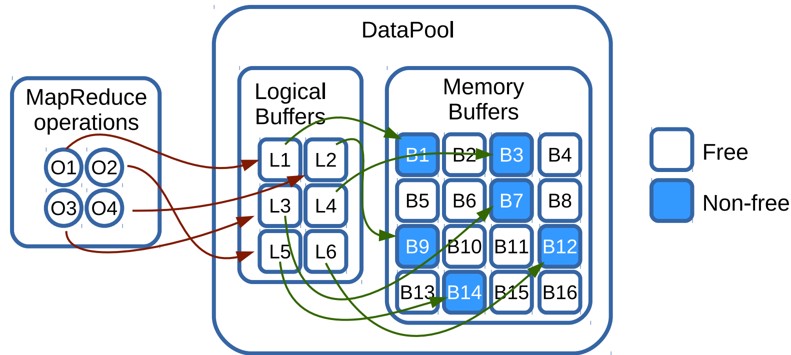


Figure 1: DataPool overview in Flame-MR-GCop

program stalls. In fact, it is one of the most common performance issues in Java applications [4]. For Big Data frameworks, this issue becomes even more important, as they execute workloads with very long execution times, creating a large number of objects and having high memory usage. For this reason, Flame-MR has been analyzed to identify means of reducing the amount of garbage collection performed, which has led to two main techniques, static memory allocation and buffered map output, explained next. These techniques have been incorporated into a modified version of Flame-MR, called Flame-MR-GCop.

3.1.1. Static memory allocation

In the original implementation of Flame-MR, MapReduce operations (e.g., map, merge) allocate memory space by requesting DataBuffers to a DataPool. Each of these requests causes a new object to be created, with its corresponding memory allocation on the heap. Once the DataBuffer is used, it is returned to the DataPool, which dereferences the DataBuffer in order to be garbage collected. This behavior involves a lot of DataBuffer object creations and collections, with their corresponding memory allocations and deallocations. Moreover, the default implementation of DataBuffers relies on primitive byte arrays to contain the data, which are initialized to a default value at the time of creation. In Flame-MR-GCop, the behavior of the DataPool has been completely redesigned, using static memory allocation in order to avoid the excessive creation of DataBuffers and to minimize GC overheads.

The basic principle of static memory allocation is that once the memory has been allocated, it is reused by several operations, without neither releasing it nor dereferencing the objects. Figure 1 depicts the new design of the DataPool, where DataBuffers are managed by means of two different classes: LogicalBuffers and MemoryBuffers. On the one hand, LogicalBuffers are associated with the data they contain, which can be in memory or in disk. On the other hand, MemoryBuffers are associated with the memory space that is allocated to them, each one containing a byte array of a fixed size (1 MB by default). LogicalBuffers reference to MemoryBuffers, using them to store the data from the operations. When a LogicalBuffer is created by an operation, it requests a MemoryBuffer to the DataPool. If the DataPool has a free MemoryBuffer available it is returned, otherwise a new one is created.

When the DataPool reaches the maximum number of buffers, it will not be able to create a new one. Instead,

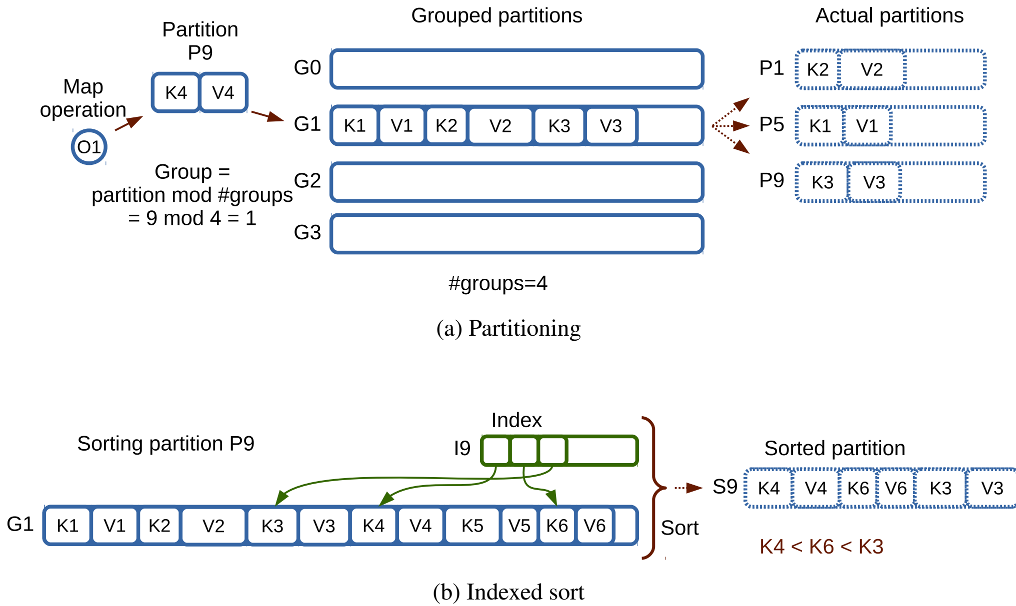
the DataPool picks a LogicalBuffer from the list and spills its corresponding MemoryBuffer to disk, giving it to the new LogicalBuffer. The former LogicalBuffer keeps a reference to the disk file, retrieving the data when an operation attempts to read its content. Finally, when a LogicalBuffer stops being used, its MemoryBuffer is put back to the DataPool. In order to minimize the number of spills and recoveries, the DataPool chooses the victim LogicalBuffer using a priority-aware schema, which uses one queue for each operation type that creates the LogicalBuffers. These queues are ordered depending on the likelihood of the LogicalBuffers to be requested back from disk. For example, LogicalBuffers created in shuffle operations will have the lowest priority, as they will have to wait until the end of the map phase to be merged. Therefore, they are the first candidates to be spilled.

In Flame-MR, most of the Worker memory is held by the DataPool, and so reusing MemoryBuffers reduces a high amount of memory allocations and deallocations. It also reduces the time taken to initialize the arrays when they are allocated, as the MemoryBuffers are reused without being reinitialized. Instead, they are lazily cleared by setting their writing position to zero. Moreover, both LogicalBuffers and MemoryBuffers use the same DataBuffer interface in order to keep a good design flexibility. MapReduce operations can therefore allocate buffers and operate the data without having specific information about where they are stored.

3.1.2. Buffered map output

One of the main objectives of Flame-MR was to minimize the redundant memory copies in operations like sort and merge. To this end, the key-value objects produced by the map operations are kept in memory until being sorted, without writing them to a temporary buffer. This behavior allows to reduce the memory copies but can lead to scalability issues with very large problem sizes due to the high amount of objects to be managed during GC, causing long stalls. In order to overcome these limitations, the writing of the map output has been improved in Flame-MR-GCop by storing binary representations of the output pairs to a temporary buffer. This avoids to keep the objects in memory and reduces the GC overhead, although it increases the number of memory copies. Furthermore, the partitioning and sorting mechanisms have been carefully studied, reassembling them for improved performance and scalability, as described next.

In Flame-MR-GCop, each mapper has a set of temporary buffers available for writing the output. When a map operation produces an output key-value pair, it is written to one of these buffers. Once a buffer is full, the output pairs are sorted and sent through the network. The number of times this event is triggered depends on how many temporary buffers are available for each mapper. Using one single buffer for all partitions is likely to cause an excessive number of sorts, as the buffer will be filled very quickly. However, having one buffer per partition is not a feasible option due to its poor scalability (the number of partitions increases with the number of computing nodes in the cluster). Therefore, the solution adopted in Flame-MR-GCop is to arrange the partitions into a set of groups, calculated as a configurable percentage of the available memory. Each group has an assigned buffer, which contains the output pairs belonging to the partitions in that group. When this buffer is full, the sorter is in charge of iterating the several partitions of the group and sorting their corresponding pairs. Figure 2a shows an example of a pair partitioning, calculating the



(a) Partitioning

(b) Indexed sort

Figure 2: Map output example

partition group number and its associated temporary buffer. The key-value pair $<K_4,V_4>$ is in partition 9, which belongs to group 1. This group contains those partitions P_i which meet the condition i mod $\#groups = 1$, which are P_1, P_5, P_9, and so on.

Each partition has an associated index that keeps track of the positions where the pairs are stored in the buffer. In the sort operation, this index is used for ordering the positions by comparing the pairs in the buffer. No data objects are created in this operation. Instead, the sort algorithm uses a binary comparison of the content of the positions. Figure 2b shows an example of a sort operation. The index (I_9) is used to access the positions that contain pairs $<K_4,V_4>$, $<K_6,V_6>$ and $<K_3,V_3>$. Then, the index is sorted according to the order of the keys, $K_4 < K_6 < K_3$. Finally, the output pairs are copied to another buffer (S_9) which will be used for sending the data to the corresponding Worker.

3.1.3. Experimental results

This section evaluates the optimizations described in Sections 3.1.1 and 3.1.2 by comparing the original version of Flame-MR and the optimized one (Flame-MR-GCop) using the Sort benchmark. As the map and reduce functions do not perform any computation over the data, the results are only affected by the overall efficiency of the MapReduce engine, which makes Sort a suitable benchmark for evaluating memory optimizations. In order to avoid repetitive results, the optimizations have not been assessed separately, thus focusing on the benefits that Flame-MR-GCop can provide as a whole. Some early experiments have been carried out in order to assess the indexed sort mechanism, which reduced up to 39% the longest Garbage Collection Time (GCT) among the Workers compared to sorting the map output buffer by creating all the objects. This provided small reductions in execution time, and so the indexed

Table 3: Sort results for Flame-MR and Flame-MR-GCop
ET: Execution Time; GCT: Garbage Collection Time

Framework	ET Min	GCT Best / Worst / Total	ET Median	GCT Best / Worst / Total	ET Max	GCT Best / Worst / Total
Flame-MR	864s	10s / 34s / 685s	1000s	12s / 36s / 782s	1454s	14s / 170s / 828s
Flame-MR-GCop	649s	3s / 7s / 119s	744s	3s / 6s / 128s	819s	3s / 5s / 124s

sorting mechanism has been used in the next experiments. Regarding the number of map output buffers, by default they occupy up to 40% of the available Worker heap size. In the experiments, the corresponding number of buffers is higher than the total number of partitions in the cluster, so each buffer has been assigned to a single partition.

Table 3 shows the experimental results. The Execution Time (ET in the table) represents the time taken by the workload to be completed. Due to the high variability between experiments, the results include the minimum, median and maximum ET. The corresponding GCT metric also shows great variability, and so several results are provided. The best and worst cases represent the Workers with the shortest and longest GCT, respectively, while the total GCT represents the sum of the GCT from all the 32 Workers (i.e., 8 slave nodes and 4 Workers per node, see Table 2). The GCT of each Worker was obtained with jstat [30]. As can be seen, the memory management improvements of Flame-MR-GCop obtain a much lower GCT (best, worst and total) compared to the previous version. This is reflected on ET, especially in the maximum case, which has an 85% reduction in total GCT and a 44% reduction in ET. Note that the worst GCT is the value that determines the delay caused by GC overheads, since the fastest Workers will have to wait for the slowest ones. The results show a clear correlation between the worst GCT and ET for Flame-MR, but not for Flame-MR-GCop. This means that the worst GCT of the improved version does not have such a great impact on ET as in the previous version.

Figure 3 presents the evolution of memory usage over time. The graphs show two scenarios: the Worker with the shortest GCT (Figures 3a and 3b) and the longest GCT (Figures 3c and 3d) for the experiment with median ET, previously shown in Table 3. The lines in these graphs depict different values related with memory behavior, including the accumulated GCT along the execution of the Worker. The JVM memory size shows the memory occupied by the Worker process, while the heap usage shows the actual size of memory that is being used by Java objects.

It can be seen that Flame-MR-GCop (Figures 3b and 3d) presents a significantly lower GCT than Flame-MR (Figures 3a and 3c), along with a more stable heap usage. In Flame-MR, the heap usage has a great variability over time, as the DataBuffers are created and dereferenced as needed. It can be observed that merge operations, which begin at 240s approximately in both scenarios, cause the highest variability in heap size, increasing the JVM size to its maximum value. This behavior is motivated by the high amount of DataBuffers that each merge operation consumes and produces, which also increases GCT, as shown in the graphs. In contrast, the heap usage of Flame-MR-GCop has a low variability, with increasing values until a certain maximum. Once this maximum is reached, the heap usage does not decrease, as the DataBuffers are kept by the DataPool in order to be reused. The reuse of DataBuffers also

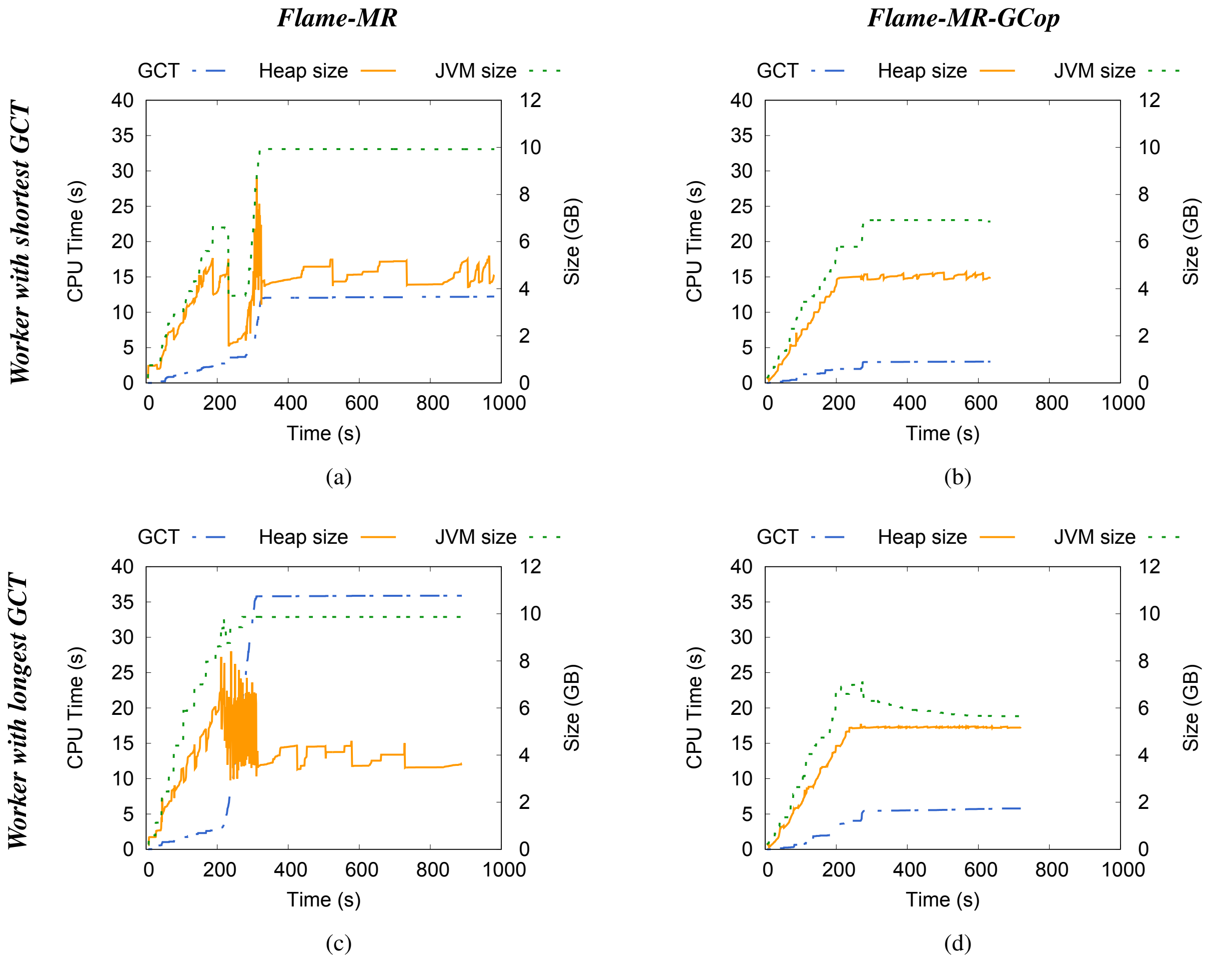


Figure 3: GCT and memory usage over time for Flame-MR and Flame-MR-GCop

reduces the GCT, making the merge operations go unnoticed in terms of memory activity compared with Flame-MR.

The results shown in this section demonstrate the impact of the GC overhead on workload performance, as well as the effectiveness of the static memory allocation and buffered map output techniques for reducing the GCT and thus the overall execution time. Moreover, the variability on heap size is also significantly reduced.

3.2. Buffer type analysis

In Java, there are two basic ways of allocating memory for new objects: on-heap and off-heap. On the one hand, on-heap memory is used by general objects and classes, being always tracked and deleted if needed by the GC. On the other hand, off-heap memory can be used for allocating buffers outside the Java heap (i.e., in native memory), which avoids copying data from heap space to native memory during OS calls (e.g., I/O operations). For storing bytes, which is the data type internally used by Flame-MR, both kinds of memory can be allocated using the ByteBuffer class provided by the JVM. ByteBuffer objects can then be used as the source and destination of I/O

system calls. The underlying memory type that is being used is encapsulated by the ByteBuffer object, using the HeapByteBuffer subclass for on-heap memory (wrapping a primitive byte array) and the DirectByteBuffer subclass for off-heap memory (wrapping memory allocated outside the heap using a malloc-like call). Although off-heap memory is out of the control of the GC (i.e., memory buffers are never moved), DirectByteBuffer objects can be garbage collected. So, the deallocation of off-heap memory is also performed during GC execution.

3.2.1. MemoryBuffer implementations

As introduced in Section 3.1.1, Flame-MR manages memory using MemoryBuffers. The underlying implementation of these buffers is abstracted by the DataBuffer interface, and so their external behavior is separated from the actual memory operations they perform. This feature allows the implementation of several kinds of MemoryBuffers that allocate memory in different ways. By default, MemoryBuffers use primitive byte arrays (i.e., *byte[]*) to store the data on the heap. Additionally, two MemoryBuffer implementations have been developed in Flame-MR-GCop, using HeapByteBuffers and DirectByteBuffers, respectively.

Although using DirectByteBuffers can potentially reduce memory copies and improve the performance of some I/O operations, it adds certain limitations to the way data can be accessed. In fact, these buffers cannot be referenced using primitive data types from Java objects stored in the heap space. Moreover, Hadoop libraries do not currently support the use of ByteBuffers for writing data to HDFS, and so output operations have to use primitive byte arrays. These issues make it difficult to determine theoretically which MemoryBuffer implementation is the most suitable. The next section analyzes the performance of the different alternatives.

3.2.2. Experimental results

The experiments to evaluate the three implementations of MemoryBuffers were conducted in a similar way to those of Section 3.1.3. The ET and GCT results of the Sort executions are shown in Table 4. Regarding ET, it can be seen that DirectByteBuffers obtain better results than the other two alternatives. Although the minimum ET is the same as with byte arrays, the median and maximum results are better. As explained in Section 3.2.1, HeapByteBuffers behave as encapsulated byte arrays, showing similar performance than these ones, plus the overhead of managing the actual ByteBuffer objects, which causes HeapByteBuffers to have a higher ET. Moreover, the results do not show any clear correlation between GCT and ET. Although DirectByteBuffers are the ones with slightly higher total GCT, the differences are almost negligible.

The graphs of Figure 4 show the accumulated GCT, heap size and JVM size of the Worker with the shortest and the longest GCT for the experiment with median ET (see Table 4). As expected, these graphs do not show any significant variation between byte arrays (Figures 4a and 4d) and HeapByteBuffers (Figures 4b and 4e), as they are using the same kind of memory in the end. However, the behavior of DirectByteBuffers (Figures 4c and 4f) greatly differs from them, showing a smaller heap size and larger JVM size. The former is caused by the off-heap allocation of DirectByteBuffers, which is obviously not reflected in the heap size. Meanwhile, the additional copies needed for

Table 4: Sort results for different buffer types in Flame-MR-GCop
ET: Execution Time; GCT: Garbage Collection Time

Buffer type	ET Min	GCT Best / Worst / Total	ET Median	GCT Best / Worst / Total	ET Max	GCT Best / Worst / Total
Byte array	649s	3s / 7s / 119s	744s	3s / 6s / 128s	819s	3s / 5s / 124s
HeapByteBuffer	681s	2s / 6s / 118s	764s	3s / 5s / 117s	855s	3s / 6s / 124s
DirectByteBuffer	649s	2s / 7s / 121s	696s	2s / 11s / 130s	779s	2s / 8s / 130s

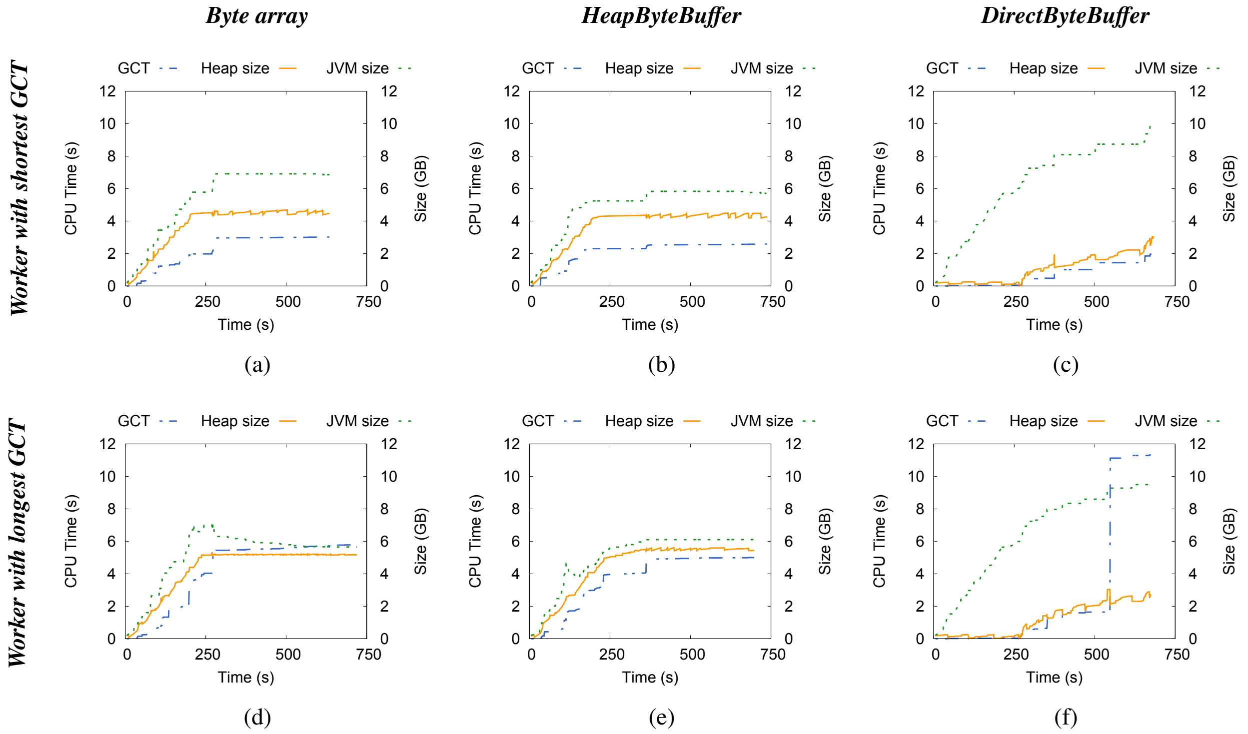


Figure 4: GCT and memory usage over time for the different DataBuffer implementations in Flame-MR-GCop

accessing the data from the heap are responsible for the larger JVM size. The use of off-heap memory also reduces the GCT in most of the computation time. However, the scenario of the Worker with the longest GCT (Figure 4f) shows a remarkable characteristic, which is a high amount of GCT consumed in a short period of time when reaching the end of the workload. This is caused by the deallocation of off-heap buffers to make room for on-heap ones, needed to write the final output to HDFS due to its API incompatibility. When the off-heap MemoryBuffers are deleted from the DataPool, the DirectByteBuffers stop being referenced, and so the GC collects them. The deallocation of off-heap memory, included in this process, has a high performance penalty, and so the GCT is affected.

The conclusion that can be extracted from these results is that off-heap buffers generally obtain a better performance than on-heap alternatives. Although some operations are more expensive when performed outside the heap (e.g., binary comparison), the improvement of I/O operations allows to get better overall results. Currently, the main

issue about using DirectByteBuffers is the lack of an API method for storing them to HDFS, which is expected to be overcome with future releases of Hadoop libraries.

3.3. Iterative support

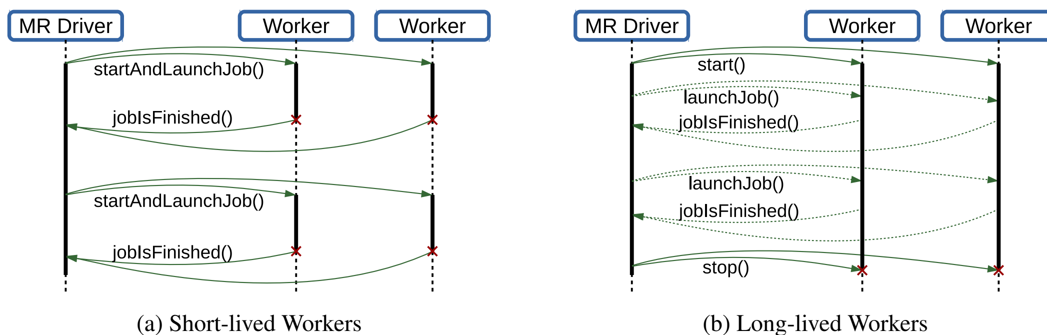
In real use cases, most MapReduce applications are composed by more than a single job, iterating several times over the input dataset to get a final result. Each MapReduce job reads the output of the previous one and generates a new dataset, until meeting a certain stop condition or reaching the maximum number of iterations. Although iterative applications are very common, MapReduce frameworks like Hadoop are not well suited for them. This is caused by the writing of intermediate results to HDFS and the high overhead of launching and finishing jobs, which leads to poor performance of the overall workload.

Like Hadoop, the versions of Flame-MR discussed in the previous sections are not oriented to iterative applications. This section focuses on how to improve the performance of these applications, avoiding the writing of intermediate results to HDFS by caching data in memory to minimize the use of disk and maximize in-memory processing. Two steps have been carried out to achieve this objective: the use of long-lived Workers and the implementation of a data cache.

3.3.1. Long-lived Workers

In each execution of a MapReduce job, Flame-MR starts one or more Workers per computing node in the cluster. When the job finishes these Workers are stopped, and hence they have to be restarted to launch the following job. In a similar way, Hadoop starts a Java process for each mapper or reducer, and so each job also involves the launching and stopping of multiple JVMs. Both approaches disable to cache intermediate results in memory between iterations, as all in-memory data are lost when the corresponding JVMs are finalized (i.e., all data have to be written to HDFS). Therefore, the first requisite for implementing a data cache in Flame-MR is to avoid stopping and restarting the Workers, reusing them through all the MapReduce jobs performed during the execution of an application. This feature also minimizes the overhead of launching new Worker processes in each job.

The proposed solution is to modify the way Workers are managed, avoiding the finalization of their JVMs when a MapReduce job is completed. Figure 5 depicts the main differences between the previous behavior of Workers, called short-lived Workers, and the new approach, long-lived Workers. On the one hand, short-lived Workers (see Figure 5a) are started for executing a certain job, being stopped after this job is finished. On the other hand, long-lived Workers (Figure 5b) are reused several times. Instead of being started for running a job, these Workers are initialized at the beginning of the MapReduce Driver (MR Driver in the figures). They keep waiting until they receive a job launch message from the driver and execute the corresponding MapReduce job. When the job finishes, they send a message back to the driver and keep waiting for more jobs to run. When the application concludes, the driver sends a message for stopping the Workers and releasing the computational resources.



(a) Short-lived Workers

(b) Long-lived Workers

Figure 5: Short-lived vs long-lived Workers

The use of long-lived Workers is mandatory to implement a data cache. It also improves the performance of iterative applications, reducing the overhead between iterations by avoiding the costly initialization of new JVM processes in each job. Furthermore, the internal structures of Flame-MR, such as DataPool and ThreadPool, are also reused along the entire application workflow, initializing them only once. Note that this includes the costly allocation and initialization of MemoryBuffers, already commented in Section 3.1.1.

3.3.2. Data cache

Long-lived Workers can be configured to use a data cache in order to avoid the writing of intermediate results to HDFS. When this feature is activated, each Worker stores its output DataBuffers to an in-memory DataStructure, which is indexed in a DataCache object. Figure 6 illustrates the behavior of the DataCache with an example of a cache hit. The output from several MapReduce jobs is tracked by associating the output path of each one (Path_i) with its corresponding DataStructure (D_i), which contains a set of LogicalBuffers (L_j). When a new job starts, the Worker reads the entries in the DataCache to check if the input path of the job is present. In this example, the input data (D_2) is read directly from the DataStructure stored in the DataCache. Otherwise, the input data has to be read from HDFS. When the job finishes, it adds its own output DataStructure (D_4) to the DataCache, making it available for future jobs. When the entire application finishes and the Worker is stopped, the contents of the DataCache are flushed to HDFS making use of the path information. After that, the dataset stored in HDFS is equivalent to the one written by a non-cached execution of the application.

The output of a MapReduce job is often accessed by the MR Driver before the entire application is completed. In fact, many iterative applications perform different kinds of operations over the intermediate results, like moving data from one path to another, deleting discarded results or reading some of them to check a stop condition. In these cases, the data cached by the Workers would be unavailable for the MR Driver, causing either errors in the driver program or incorrect results. The behavior of HDFS calls must thus be modified in order to ensure the same results as in non-cached jobs. First, jobs create empty output directories when finished, so the HDFS state from the point of view of the MR Driver remains the same. The HDFS calls are then monitored during execution to modify the content

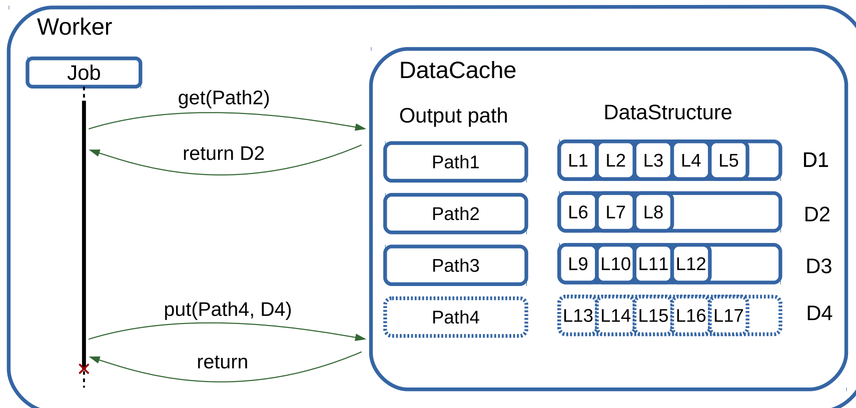


Figure 6: Data cache example

of the DataCaches accordingly. When the MR Driver attempts to delete a path that is cached, its entry is removed and the DataBuffers are released. When a path is moved, the corresponding entries change the associated path, with no writings to HDFS. Finally, when the driver attempts to read a dataset, the Workers flush the content of the cached DataStructures to HDFS, filling the empty output directories. After the data are available in HDFS, the corresponding DataCache entries are removed and the MR Driver can then access data in the standard way. This behavior ensures that the data of the output directory are always entirely in memory or in HDFS, preventing any inconsistency in subsequent jobs. However, in some cases, a MapReduce job can process data from HDFS and the DataCache simultaneously, when the input is composed of several paths stored in different places. As data are independently stored either in DataBuffers or HDFS blocks, they can always be read and processed in parallel. Note also that HDFS generally keeps several replicas of each data block as configured by the user, which ensures the reliability of the intermediate results. Our current implementation of the DataCache does not perform any replication, and so if a Worker is torn down, it would mean the loss of all its intermediate results. Although the reliability of the final results is ensured after writing them to HDFS, we are currently designing new fault tolerance mechanisms to avoid data loss in case of Worker breakdown, but reducing disk and network overheads.

The approach described has some similarities with the one implemented in M3R [22], which uses a key-value pair cache through the execution of in-memory iterative jobs. However, our implementation differs in the way data are kept in memory, using instead a binary format to store the data in DataBuffers and thus reducing the overhead incurred by the GC tracking. Futhermore, the data size can be higher than the available memory space, in which case some of the buffers will be spilled to disk, while M3R can only work with in-memory data and thus with a maximum data size.

The optimizations explained in this section, long-lived Workers and the data cache, have been integrated in Flame-MR-GCop, resulting in a new version called Flame-MR-It. The following section analyzes the performance improvement obtained with this iterative-aware version.

Table 5: Execution times for PageRank

	Flame-MR	Flame-MR-GCop	Flame-MR-It-NoCache	Flame-MR-It-Cache
PageRank	1895s	1429s	1176s	1060s

3.3.3. Experimental results

This section analyzes the performance and resource efficiency of the iterative application PageRank. Although other iterative workloads have also been tested, the results did not differ significantly from the ones obtained with PageRank, and so they were not included in this section. Table 5 shows the median execution time of the different versions of Flame-MR. In order to analyze the performance improvement obtained by the use of the data cache, Flame-MR-It has been executed with and without activating the cache (Flame-MR-It-Cache and Flame-MR-It-NoCache, respectively). The results show that both configurations of Flame-MR-It reduce significantly the execution time of PageRank with respect to the previous versions. Using Flame-MR-GCop as baseline, Flame-MR-It-NoCache reduces the execution time by 18%, while Flame-MR-It-Cache decreases it by 26%. Hence, most part of the obtained improvement is because of the use of long-lived Workers, due to the reasons pointed out at the end of Section 3.3.1. This reveals that for this application the initialization of the Workers have a significant impact on performance, being even more significant than the writing of the intermediate results to HDFS (only 38 GB per iteration in this case), avoided by the use of the data cache.

Figure 7 depicts the resource utilization statistics of Flame-MR and both versions of Flame-MR-It. In terms of CPU usage, both Flame-MR-It versions show lower values along the entire computation of the workload. This is caused by the GC optimizations explained in Section 3.1, as well as by the reduction of the initialization overhead between iterations. Furthermore, the activation of the data cache has a direct effect on CPU usage, almost eliminating the CPU time waiting for I/O. Regarding memory usage[1], Flame-MR-It-NoCache takes further advantage of the memory optimizations, avoiding the maximum and minimum peaks of Flame-MR by means of the static DataBuffer allocation (see Section 3.1.1). The use of long-lived Workers also improves memory usage by avoiding the release of the entire JVM memory space each time a job is finished. Activating the data cache also contributes to a more stable memory usage, as it avoids the increase of the memory used by the OS buffer cache.

Both Flame-MR and Flame-MR-It-NoCache show high disk utilization in each iteration of PageRank, reaching 100% values during several intervals of the workload. Meanwhile, Flame-MR-It-Cache shows almost no disk utilization, as the intermediate results are neither written to nor read from HDFS. Regarding network traffic, both Flame-MR and Flame-MR-It-NoCache show higher values than Flame-MR-It-Cache. This is caused by the sending of data blocks through the network when they are replicated according to the configured replication factor (3, as can be seen in Table 2). As Flame-MR-It-Cache does not write any intermediate results to HDFS, it only incurs the traffic belonging to the shuffle phase. In conclusion, the use of the data cache not only improves performance but also increases resource

[1] Note that the “Cached” label refers to the OS buffer cache, not to the data cache of Flame-MR-It-Cache

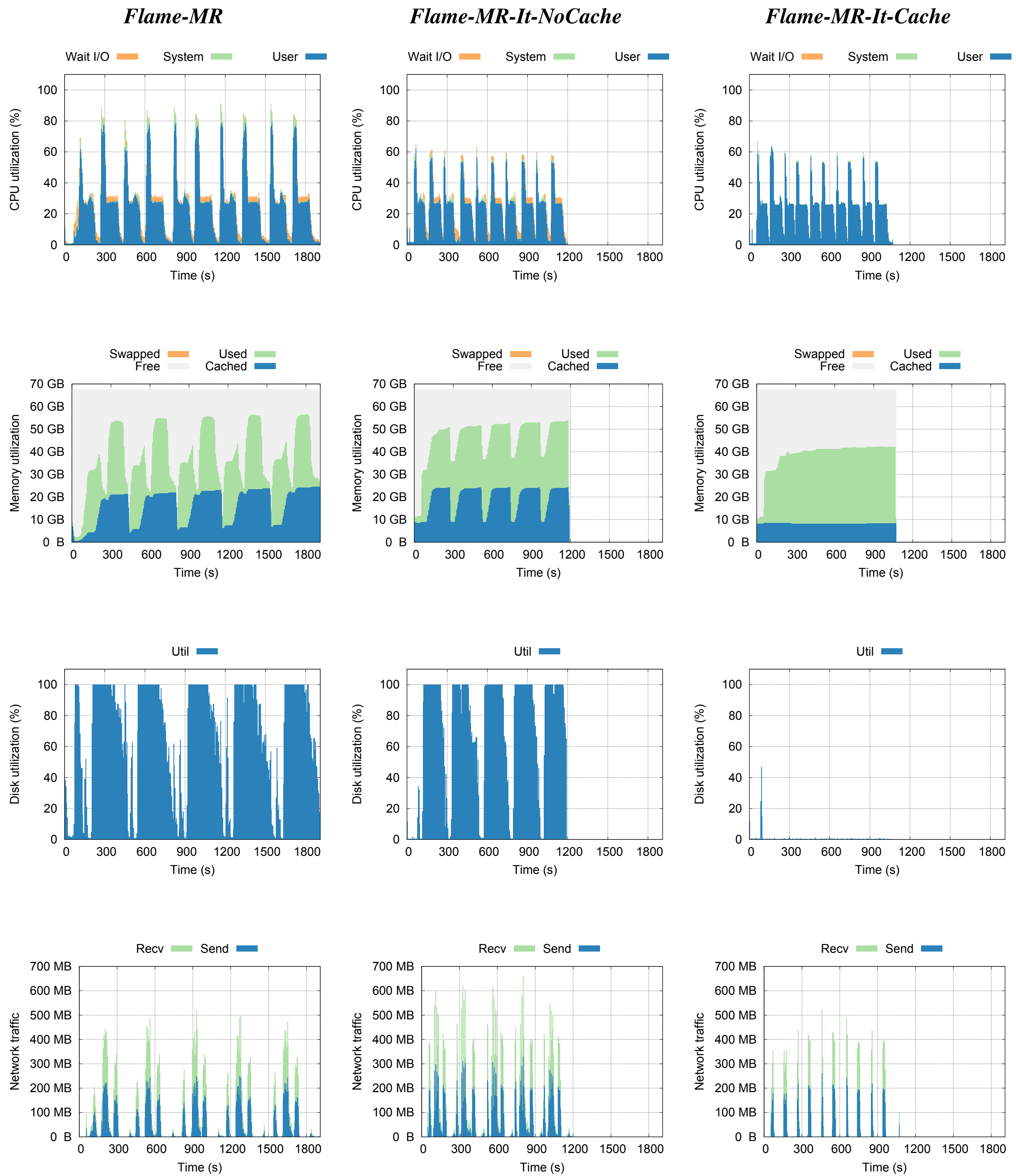


Figure 7: Resource utilization statistics of Flame-MR, Flame-MR-It-NoCache and Flame-MR-It-Cache

efficiency.

4. Putting it all together: performance comparison with Hadoop and Spark

This section presents an experimental evaluation of the memory-enhanced and iterative-aware version of Flame-MR (i.e., Flame-MR-It) compared to popular Big Data processing frameworks. First, Section 4.1 compares Flame-MR-It with the de-facto standard MapReduce implementation, Apache Hadoop [1], assessing the impact of managing memory in a more efficient way when executing representative MapReduce workloads at a sufficiently large scale. This evaluation takes into account both performance and resource utilization metrics, also including the results for the original version of Flame-MR in order to determine the improvement achieved by all the optimizations presented in Section 3. Then, Section 4.2 analyzes the performance benefits of using Flame-MR-It compared to a state-of-the-art in-memory framework, Apache Spark [2]. Note that both frameworks are oriented to leverage memory resources in order to improve the performance of Hadoop. Hence, comparing them is useful to determine the performance gain that Flame-MR-It provides, unlike Spark, without modifying the source code of existing MapReduce applications.

As the Pluton cluster has a limited number of computing nodes, the experiments have been carried out in a public cloud infrastructure, Amazon EC2 [6], which is the most popular Infrastructure as a Service (IaaS) platform. In order to evaluate the behavior of the frameworks under different memory configurations, two Amazon EC2 instance types have been used: c3.4xlarge and i2.4xlarge, which were allocated in the US East (North Virginia) region. Table 6 contains the main hardware and software characteristics of both instances, as advertised by Amazon. As can be seen, the memory size of c3.4xlarge is 30 GB, while i2.4xlarge has 122 GB. The wide difference in terms of memory between these instances is of great interest to study the performance of the frameworks when running in a more memory-constrained system (i.e., c3.4xlarge) compared to i2.4xlarge instances, allowing to analyze the impact of memory management under different system configurations. Furthermore, i2.4xlarge instances also provide more local storage: 4 SSD disks of 800 GB vs. 2 SSD disks of 160 GB for c3.4xlarge. However, note that the CPU speed of c3.4xlarge is slightly higher, which may improve the performance of CPU-bound workloads. The experiments were run on a 33-instance cluster (i.e., 1 master and 32 slaves).

Several benchmarks have been analyzed by using BDEv [27]: WordCount, Sort, PageRank, Connected Components and K-Means. Sort and PageRank were already introduced in Section 3.1. WordCount is a CPU-bound benchmark included in the Hadoop distribution that reads an input dataset and computes the number of times each word appears. Connected Components is, as PageRank, an iterative application also provided by Pegasus [25] that explores a graph to determine its subnets. K-Means is an iterative clustering algorithm that partitions a set of samples into K clusters. Mahout [31] provides the MapReduce implementation of K-Means along with its input dataset generator, GenKMeansDataSet. The Spark implementation of the benchmarks has been derived from its code examples except in the case of Connected Components and K-Means, which use the graph processing engine GraphX and the machine learning library MLlib, respectively. In the experiments, WordCount and Sort processed a 500 GB dataset,

Table 6: Characteristics of the Amazon EC2 instances

Hardware configuration	c3.4xlarge	i2.4xlarge
CPU	2 × Intel Xeon E5-2680 v2 Ivy Bridge-EP	2 × Intel Xeon E5-2670 v2 Ivy Bridge-EP
CPU Speed/Turbo	2.80 GHz/3.60 GHz	2.50 GHz/3.30 GHz
#Cores	16	16
Memory	30 GB DDR3 1600 MHz	122 GB DDR3 1600 MHz
Disk	2 × 160 GB SSD	4 × 800 GB SSD
Networks	Gigabit Ethernet	Gigabit Ethernet

Software configuration	
OS version	Amazon Linux AMI 2016.09
Kernel	4.4.19-29.55.amzn1.x86_64
Java	OpenJDK 1.7.0_111
Scala	2.10.6

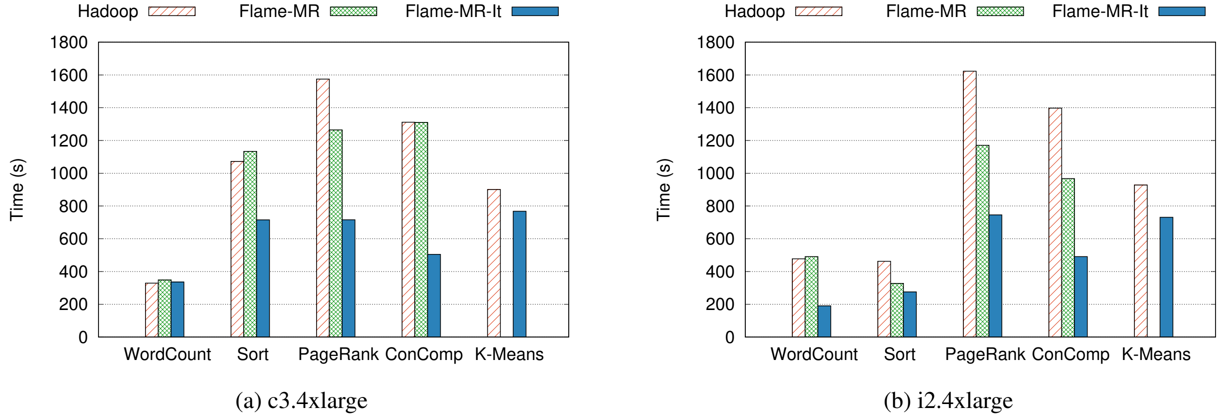
Table 7: Configuration of the frameworks

Hadoop			Flame-MR / Flame-MR-It		
	c3.4xlarge	i2.4xlarge		c3.4xlarge	i2.4xlarge
Mappers per node	8	8	Workers per node	4	4
Reducers per node	8	8	ThreadPool size	4	4
Mapper/Reducer heap size	1.7 GB	6.33 GB	Worker heap size	5.4 GB	22.8 GB
IO sort MB	380 MB	1.58 GB	DataPool size	3.3 GB	15.2 GB
Shuffle parallel copies	20	20	DataBuffer size	512 KB / 128 KB	1 MB / 256 KB
IO sort spill percent	80%	80%			

Spark			HDFS	
	c3.4xlarge	i2.4xlarge		c3.4xlarge / i2.4xlarge
Executor heap size	23.8 GB	101.3 GB	HDFS block size	128 MB
Executors per node	1	1	Replication factor	3
Executor cores	16	16		

while PageRank and Connected Components performed 5 iterations over a 40 GB dataset (60 Mpages). K-Means processed an input dataset of 130 GB (3 Gsamples), using 10 clusters and performing a maximum of 5 iterations. The metric shown in the graphs is the mean value of 10 measurements for each experiment, clearing the OS buffer cache between each execution.

Finally, the evaluated frameworks have been Hadoop 2.7.2, Spark 1.6.3, the original version of Flame-MR and its optimized version, Flame-MR-It. The latter has been configured to use the data cache in the iterative benchmarks (PageRank, Connected Components and K-Means), and byte arrays as the underlying implementation for DataBuffers. In order to provide the best results for each framework, their configuration, shown in Table 7, has been carefully



(a) c3.4xlarge

(b) i2.4xlarge

Figure 8: Execution times of Hadoop, Flame-MR and Flame-MR-It in Amazon EC2

adapted to the characteristics of the instances, adjusting some of the parameters (e.g., DataBuffer size) by experimental tuning.

4.1. Comparison with Hadoop

Figures 8a and 8b show the execution times when using c3.4xlarge and i2.4xlarge instances, respectively. Note that the results of K-Means with Flame-MR are not included as this previous version did not provide support for Mahout workloads. As can be observed, Flame-MR-It significantly outperforms Hadoop with an average improvement of 32% and 48% in c3.4xlarge and i2.4xlarge, respectively. These results confirm that Flame-MR-It presents a significantly better performance than Hadoop, especially when using large memory sizes as the improvements are higher as the memory of the instances increases. The only case in which Hadoop obtains a similar performance to Flame-MR-It is when running WordCount in c3.4xlarge. This is caused by the high amount of Java objects created by the user-defined map function, which affects the performance in memory-constrained systems, as the GC has to perform a lot of collections to make room for new objects. The results also show the significant performance improvement of Flame-MR-It over Flame-MR, demonstrating the effectiveness of the addressed in-memory techniques.

Note that the choice of the instance type has a significant impact on the performance of the frameworks. The benchmarks that are influenced the most are WordCount and Sort. In fact, Hadoop and Flame-MR obtain lower execution times for WordCount in c3.4xlarge than in i2.4xlarge. This means that, for this benchmark, both frameworks are clearly constrained by the CPU power, and the better memory and disk features of i2.4xlarge do not compensate for its lower CPU speed (see Table 6). However, Flame-MR-It is not affected by this problem, achieving for WordCount better performance in i2.4xlarge than in c3.4xlarge. This indicates that Flame-MR-It obtains a higher benefit than Hadoop from the use of large memory sizes, without being penalized by the CPU speed. Regarding Sort, it is the benchmark that shows the most important benefit from using i2.4xlarge. Sort is an I/O-bound workload, so this improvement can be clearly attributed to the higher number of disks provided by i2.4xlarge (4 disks), which improves

the bandwidth of I/O operations, along with more memory space (122 GB), allowing the frameworks to retain more data in memory without spilling them to disk. As in the case of WordCount, Flame-MR-It benefits more from the use of i2.4xlarge instances, which means that, compared with Hadoop, the Flame-MR architecture together with the in-memory techniques addressed in Section 3.1 are more suited to systems with larger memory sizes.

Finally, PageRank, Connected Components (ConComp in the figure) and K-Means are iterative applications, and so their performance is determined by the behavior within each iteration and between iterations. Hence, Flame-MR-It not only improves the execution time of each iteration due to its memory optimizations, but also reduces the overhead between iterations (e.g., launching of Worker processes) by means of the techniques explained in Section 3.3: reusing the Worker processes and avoiding to write intermediate results to HDFS by caching data in memory.

The resource utilization of Hadoop and Flame-MR-It can be useful to analyze their behavior in the c3.4xlarge and i2.4xlarge instances. As previously commented, Sort was the benchmark which showed wider performance differences between both instance types. Therefore, it has been selected for analyzing the resource utilization of both frameworks, calculated as the average values among the slave nodes during the experiment with the median execution time. The disk and memory utilization results are shown in Figures 9 and 10, respectively.

As can be observed in Figure 9, the disk constitutes the main performance bottleneck in c3.4xlarge (see Figures 9a and 9b), while in i2.4xlarge (Figures 9c and 9d) the larger memory size and higher number of disks alleviate the load per disk. This, in turn, accelerates the completion of the benchmark by 2.32× and 2.60× for Hadoop and Flame-MR-It, respectively. In c3.4xlarge, Flame-MR-It presents lower disk utilization than Hadoop along the entire execution thanks to its better in-memory computing capabilities.

Regarding memory utilization, Figures 10a and 10c show that Hadoop heavily relies on the OS buffer cache to accelerate I/O operations, which matches with the high disk utilization values commented before. Meanwhile, Flame-MR-It (Figures 10b and 10d) keeps lower buffer cache values by avoiding disk access as much as possible and retaining more data in the Workers memory. This feature allows Flame-MR-It to have more available memory space to be used to maximize in-memory data processing even in memory-constrained systems like c3.4xlarge instances.

4.2. *Comparison with Spark*

Figure 11 shows the performance results of Hadoop, Spark and Flame-MR-It. As can be seen, the best performer depends on the workload being executed. On the one hand, WordCount obtains the lowest execution times with Spark. This is due to the better suitability of the in-memory data structures of Spark to CPU-bound workloads like this one, combined with the use of efficient data processing operators like the *reduceByKey()* function. On the other hand, Flame-MR-It obtains the best results for Sort, mainly because of the great data sorting capabilities of the MapReduce model together with the optimized memory and GC management for large datasets provided by Flame-MR-It. Regarding iterative workloads, PageRank, Connected Components and K-Means also differ on the best framework to use. The in-memory optimizations of Flame-MR-It prove to be highly efficient for PageRank, reducing the execution time of Spark by 35% and 20% in c3.4xlarge and i2.4xlarge, respectively. However, Spark obtains

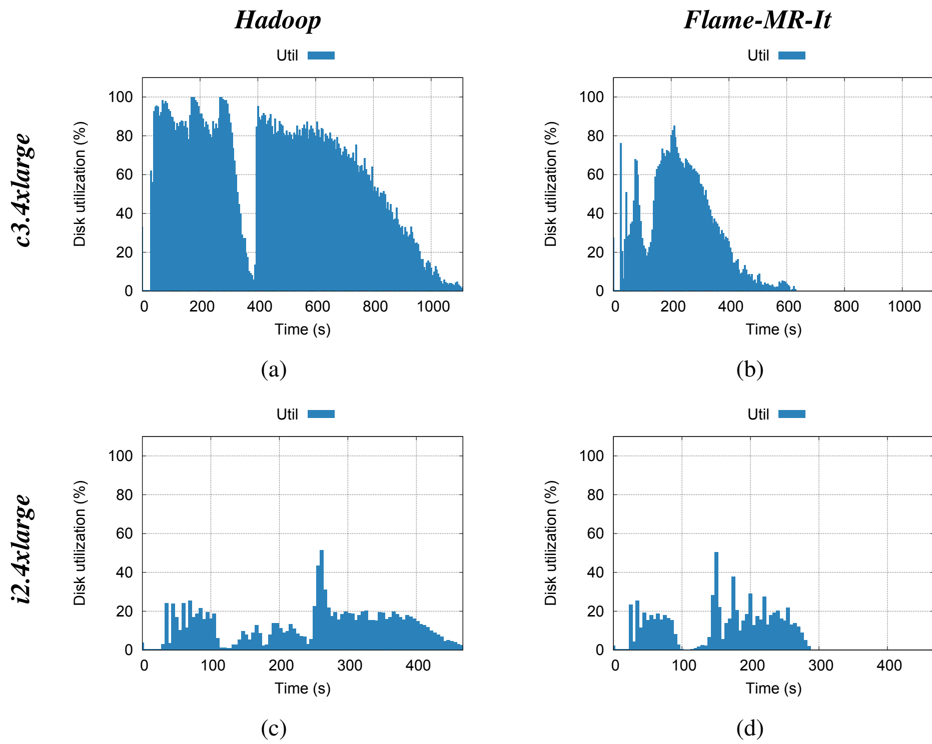


Figure 9: Disk utilization of Hadoop and Flame-MR-It for Sort

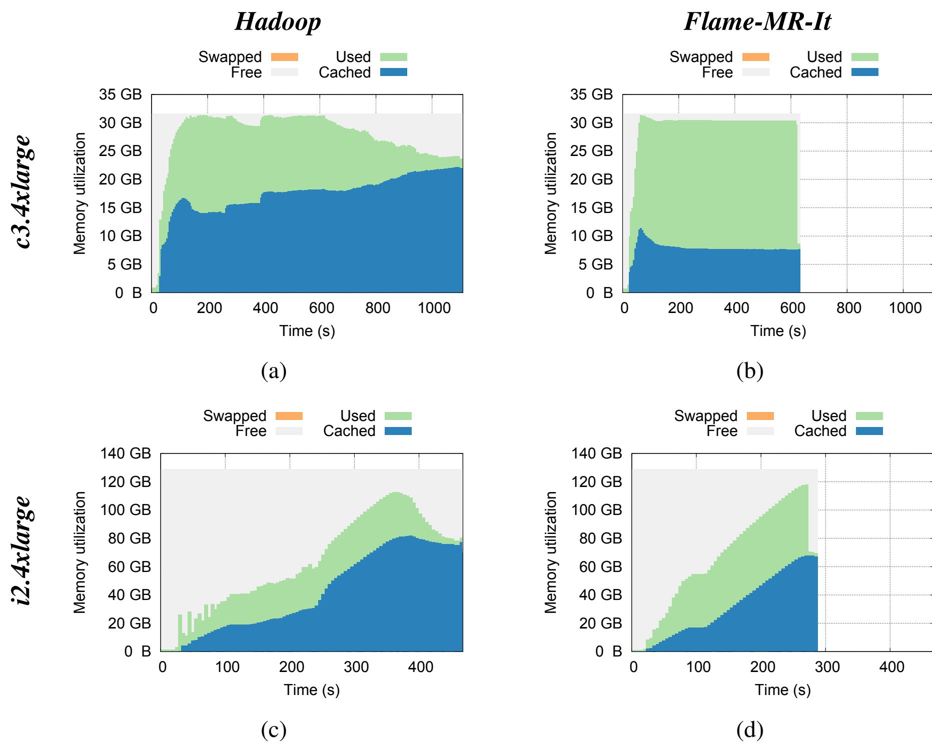
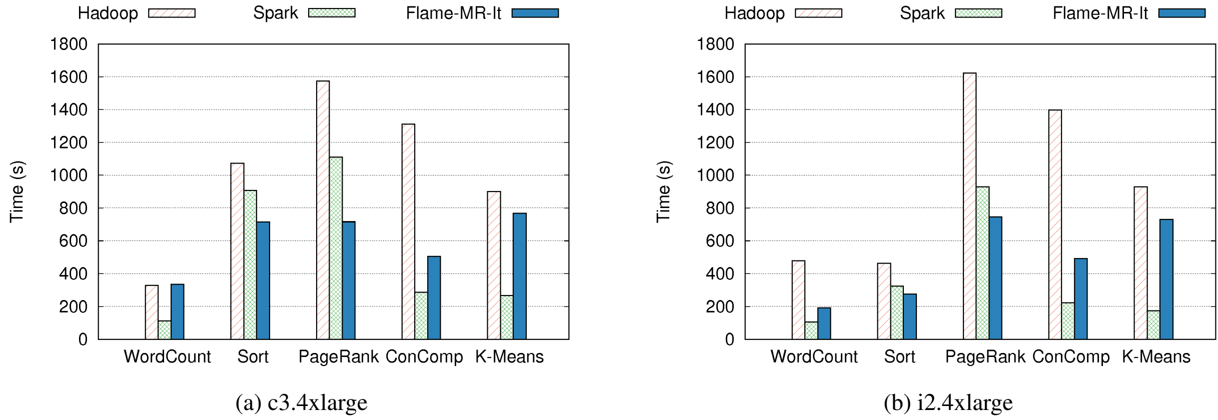


Figure 10: Memory utilization of Hadoop and Flame-MR-It for Sort



(a) c3.4xlarge

(b) i2.4xlarge

Figure 11: Execution times of Hadoop, Spark and Flame-MR-It in Amazon EC2

better performance than Flame-MR-It for Connected Components and K-Means. This is due to the use of its built-in graph processing engine, GraphX, and its machine learning library, MLlib, that fully leverage the in-memory data operators of Spark. Note that the Spark implementation of these iterative workloads can use in-memory data to check convergence at the end of each iteration, whereas the Hadoop implementation used by Flame-MR must read the data from HDFS to perform this task, thus incurring additional network and disk overhead in the MR Driver.

In order to provide more information about the performance differences of Spark and Flame-MR-It, their resource utilization has been analyzed. The results of PageRank and Connected Components are especially relevant because of the wide performance differences they show, taking into account that they process the same input dataset using different graph algorithms. Figures 12 and 13 depict the CPU utilization and network traffic of PageRank and Connected Components, respectively, for i2.4xlarge instances. Although disk and memory utilization have also been analyzed, the disk traffic is almost negligible during the entire execution of both frameworks (below 14%). The main difference in the memory utilization is that Spark consumes 62% and 25% more memory than Flame-MR-It for PageRank and Connected Components, respectively. Apart from that, both frameworks show the same pattern that does not bring any meaningful insight. Therefore, disk and memory utilization graphs are not shown in this section.

The iterative behavior of both Spark and Flame-MR-It for PageRank can be clearly seen in Figure 12. The several CPU utilization peaks correspond with the jobs performed during the workload (see Figures 12a and 12b). Network traffic also shows similar peaks of activity that belong to the data shuffling within each job (Figures 12c and 12d). Note that Flame-MR-It shows a gap without activity at the beginning of the workload (Figure 12b), which is due to the data initialization performed by the MR Driver in PageRank. Regarding Connected Components, it can be seen that the iterative behavior of the workload is present in Flame-MR-It (Figures 13b and 13d) but is not so clear in Spark (Figures 13a and 13c). As previously commented, the optimized GraphX library leverages the Spark operations in a more efficient way than the MapReduce counterpart, thus benefiting from improved operation scheduling and in turn better performance. Another remark is that the best performer is the one with the highest network traffic: Flame-MR-It

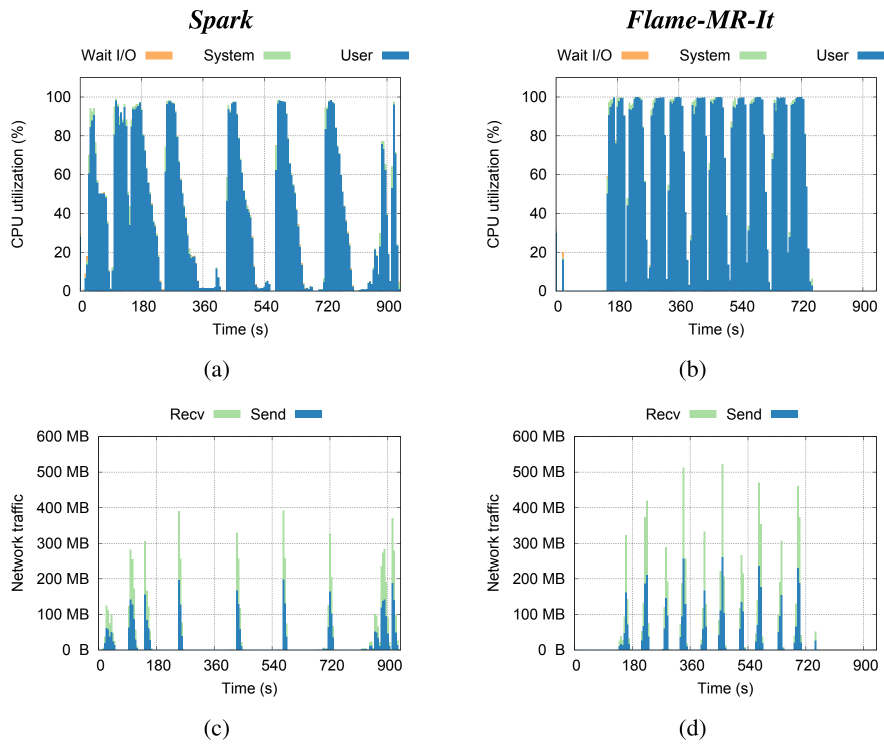


Figure 12: CPU utilization and network traffic for PageRank in i2.4xlarge

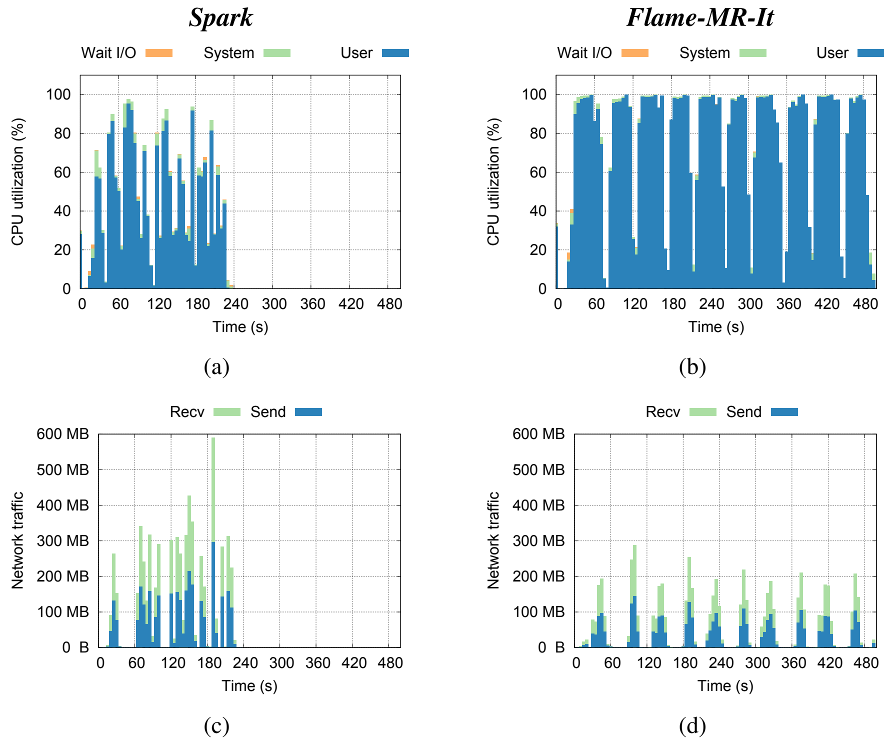


Figure 13: CPU utilization and network traffic for Connected Components in i2.4xlarge

for PageRank (Figure 12d) and Spark for Connected Components (Figure 13c). Therefore, network bandwidth is not a performance bottleneck for any of these workloads.

As a summary, the average performance improvement that Spark and Flame-MR-It obtain, compared to Hadoop, is 63% and 48%, respectively, when using i2.4xlarge instances. These results confirm the overall efficiency of our memory optimizations, as Flame-MR-It can transparently improve the performance of MapReduce workloads, while providing competitive results compared to Spark and also using less memory. Using one or another to improve the performance of an existing MapReduce application would depend on the particular workload characterization and the willingness of the user to modify the source code (if available).

5. Conclusions and future work

Future systems are expected to have increasing memory sizes, which can be challenging to current data processing frameworks. This paper has analyzed in depth the impact of memory efficiency on the performance of the Hadoop-compatible Flame-MR framework, presenting several memory optimization techniques that have been implemented and evaluated. The obtained results have shown that these techniques can reduce the amount of object allocations and deallocations, decreasing GC overheads and the overall execution times by 85% and 44%, respectively. Moreover, several memory buffer implementations have been analyzed, showing that direct byte buffers can improve the performance of I/O-bound operations. Finally, the performance of iterative workloads was improved by caching intermediate data and reusing Worker processes to avoid unnecessary writes to HDFS, reducing execution times by 26%.

The optimized implementation of Flame-MR has been compared with Hadoop on different Amazon EC2 instances, obtaining lower execution times for all benchmarks, with a maximum reduction of 65%. These performance differences become wider when using larger memory sizes. Therefore, Flame-MR is able to provide better in-memory computing capabilities than Hadoop, making it more suited for future computing systems with larger memories. Compared to representative in-memory frameworks like Spark, Flame-MR provides very competitive performance without modifying the source code of the applications. The latest version Flame-MR 1.0 includes the optimizations addressed in this paper, being publicly available at `http://flamemr.des.udc.es`.

In the near future, we plan to continue improving the efficiency of Flame-MR by leveraging other resources typically available in high-performance systems, like SSD disks or Remote Direct Memory Access (RDMA) networks.

Acknowledgments

This work was supported by the Ministry of Economy, Industry and Competitiveness of Spain and FEDER funds of the European Union (Project TIN2016-75845-P, AEI/FEDER/EU), by the FPU Program of the Ministry of Education (grant FPU14/02805), and by an Amazon Web Services (AWS) LLC research grant.

References

[1] Apache Hadoop, http://hadoop.apache.org/, [Last visited: March 2018].

[2] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, I. Stoica, Apache Spark: a unified engine for Big Data processing, Communications of the ACM 59 (11) (2016) 56–65.

[3] Apache Flink: scalable batch and stream data processing, http://flink.apache.org/, [Last visited: March 2018].

[4] M. Hertz, E. D. Berger, Quantifying the performance of garbage collection vs. explicit memory management, in: Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'05), San Diego, CA, USA, 2005, pp. 313–326.

[5] J. Veiga, R. R. Expósito, G. L. Taboada, J. Touriño, Flame-MR: an event-driven architecture for MapReduce applications, Future Generation Computer Systems 65 (2016) 46–56.

[6] Amazon Web Services Inc. Amazon Elastic Compute Cloud (Amazon EC2), https://aws.amazon.com/ec2/, [Last visited: March 2018].

[7] K. Shvachko, H. Kuang, S. Radia, R. Chansler, The Hadoop Distributed File System, in: Proceedings of the IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST'2010), Incline Village, NV, USA, 2010, pp. 1–10.

[8] J. Dean, S. Ghemawat, MapReduce: simplified data processing on large clusters, Communications of the ACM 51 (1) (2008) 107–113.

[9] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, E. Baldeschwieler, Apache Hadoop YARN: Yet Another Resource Negotiator, in: Proceedings of the 4th Annual Symposium on Cloud Computing (SOCC'13), Santa Clara, CA, USA, 2013, pp. 5:1–5:16.

[10] S. Sumimoto, Y. Ajima, K. Saga, T. Nose, N. Shida, T. Nanri, The design of advanced communication to reduce memory usage for Exascale systems, in: Proceedings of the 12th International Meeting on High Performance Computing for Computational Science (VECPAR'16), Porto, Portugal, 2016, pp. 149–161.

[11] S. Markidis, I. B. Peng, J. L. Träff, V. B. Antoine Rougier, R. Machado, M. Rahn, A. Hart, D. Holmes, M. Bull, E. Laure, The EPiGRAM project: preparing parallel programming models for Exascale, in: Proceedings of the Workshop on Exascale Multi/Many Core Computing Systems (E-MuCoCoS), Frankfurt, Germany, 2016, pp. 56–68.

[12] M. Dimitrov, K. Kumar, P. Lu, V. Viswanathan, T. Willhalm, Memory system characterization of Big Data workloads, in: Proceedings of the IEEE International Conference on Big Data (IEEE BigData 2013), Santa Clara, CA, USA, 2013, pp. 15–22.

[13] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Anthony, H. Liu, R. Murthy, Hive - a Petabyte scale data warehouse using Hadoop, in: Proceedings of the 26th IEEE International Conference on Data Engineering (ICDE 2010), Long Beach, CA, USA, 2010, pp. 996–1005.

[14] J. Park, M. Han, W. Baek, Quantifying the performance impact of large pages on in-memory Big-Data workloads, in: Proceedings of the 2016 IEEE International Symposium on Workload Characterization (IISWC'16), Providence, RI, USA, 2016, pp. 1–10.

[15] T. Yoo, M. Yim, I. Jeong, Y. Lee, S.-T. Chun, Performance evaluation of in-memory computing on scale-up and scale-out cluster, in: Proceedings of the 8th International Conference on Ubiquitous and Future Networks (ICUFN 2016), Vienna, Austria, 2016, pp. 456–461.

[16] P. Xuan, W. B. Ligon, P. K. Srimani, R. Ge, F. Luo, Accelerating Big Data analytics on HPC clusters using two-level storage, Parallel Computing 61 (2017) 18–34.

[17] I. S. Choi, W. Yang, Y.-S. Kee, Early experience with optimizing I/O performance using high-performance SSDs for in-memory cluster computing, in: Proceedings of the IEEE International Conference on Big Data (IEEE BigData 2015), Santa Clara, CA, USA, 2015, pp. 1073–1083.

[18] K. Nguyen, L. Fang, G. H. Xu, B. Demsky, S. Lu, S. Alamian, O. Mutlu, Yak: a high-performance Big-Data-friendly Garbage Collector, in: Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16), Savannah, GA, USA, 2016, pp. 349–365.

[19] L. Lu, X. Shi, Y. Zhou, X. Zhang, H. Jin, C. Pei, L. He, Y. Geng, Lifetime-based memory management for distributed data processing systems, Proceedings of the Very Large Data Base (VLDB) Endowment 9 (12) (2016) 936–947.

[20] D. Yan, X.-S. Yin, C. Lian, X. Zhong, X. Zhou, G.-S. Wu, Using memory in the right way to accelerate Big Data processing, Journal of Computer Science and Technology 30 (1) (2015) 30–41.

[21] D. Yang, X. Zhong, D. Yan, F. Dai, X. Yin, C. Lian, Z. Zhu, W. Jiang, G. Wu, NativeTask: a Hadoop compatible framework for high performance, in: Proceedings of the IEEE International Conference on Big Data (IEEE BigData 2013), Santa Clara, CA, USA, 2013, pp. 94–101.

[22] A. Shinnar, D. Cunningham, V. Saraswat, B. Herta, M3R: increased performance for in-memory Hadoop jobs, Proceedings of the Very Large Data Base (VLDB) Endowment 5 (12) (2012) 1736–1747.

[23] Y. Zhang, Q. Gao, L. Gao, C. Wang, iMapReduce: a distributed computing framework for iterative computation, Journal of Grid Computing 10 (1) (2012) 47–68.

[24] R. Gu, X. Yang, J. Yan, Y. Sun, B. Wang, C. Yuan, Y. Huang, SHadoop: improving MapReduce performance by optimizing job execution mechanism in Hadoop clusters, Journal of Parallel and Distributed Computing 74 (3) (2014) 2166–2179.

[25] U. Kang, C. E. Tsourakakis, C. Faloutsos, PEGASUS: a peta-scale graph mining system implementation and observations, in: Proceedings of the 9th IEEE International Conference on Data Mining (ICDM'09), Miami, FL, USA, 2009, pp. 229–238.

[26] S. Huang, J. Huang, J. Dai, T. Xie, B. Huang, The HiBench benchmark suite: characterization of the MapReduce-based data analysis, in: Proceedings of the 26th IEEE International Conference on Data Engineering Workshops (ICDEW'10), Long Beach, CA, USA, 2010, pp. 41–51.

[27] J. Veiga, R. R. Expósito, G. L. Taboada, J. Touriño, MREv: an automatic MapReduce Evaluation tool for Big Data workloads, in: Proceedings of the International Conference on Computational Science (ICCS'15), Reykjavík, Iceland, 2015, pp. 80–89.

[28] L. Gidra, G. Thomas, J. Sopena, M. Shapiro, Assessing the scalability of garbage collectors on many cores, in: Proceedings of the 6th Workshop on Programming Languages and Operating Systems (PLOS'11), Cascais, Portugal, 2011, pp. 7:1–7:5.

[29] Sun Microsystems, Memory management in the Java HotSpot™ Virtual Machine, http://www.oracle.com/technetwork/articles/java/index-jsp-140228.html, [Last visited: March 2018].

[30] The jstat utility, https://docs.oracle.com/javase/8/docs/technotes/tools/unix/jstat.html, [Last visited: March 2018].

[31] Apache Mahout, Scalable machine learning and data mining, http://mahout.apache.org/, [Last visited: March 2018].