Zhao P, Ding C, Liu L *et al.* Cacheap: Portable and collaborative I/O optimization for graph processing. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 34(3): 690–706 May 2019. DOI 10.1007/s11390-019-1936-6

Cacheap: Portable and Collaborative I/O Optimization for Graph Processing

Peng Zhao[1,2], *Student Member*, *CCF*, Chen Ding[3], *Member*, *ACM*, *IEEE*, Lei Liu[1,*], *Member*, *CCF*, Jiping Yu[4] Wentao Han[4], *Member*, *CCF*, *ACM*, *IEEE*, and Xiao-Bing Feng[1,2], *Member*, *CCF*, *ACM*, *IEEE*

[1] *State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences Beijing 100190, China*

[2] *University of Chinese Academy of Sciences, Beijing 100049, China*

[3] *Department of Computer Science, University of Rochester, Rochester 14623, U.S.A.*

[4] *Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China*

E-mail: zhaopeng@ict.ac.cn; cding@cs.rochester.edu; liulei@ict.ac.cn; yujp15@mails.tsinghua.edu.cn hanwentao@tsinghua.edu.cn; fxb@ict.ac.cn

Received May 9, 2018; revised March 19, 2019.

Abstract Increasingly there is a need to process graphs that are larger than the available memory on today's machines. Many systems have been developed with graph representations that are efficient and compact for out-of-core processing. A necessary task in these systems is memory management. This paper presents a system called Cacheap which automatically and efficiently manages the available memory to maximize the speed of graph processing, minimize the amount of disk access, and maximize the utilization of memory for graph data. It has a simple interface that can be easily adopted by existing graph engines. The paper describes the new system, uses it in recent graph engines, and demonstrates its integer factor improvements in the speed of large-scale graph processing.

Keywords out-of-core graph processing system, I/O optimization, memory cache, graph analytics, locality

1 Introduction

There have been increasing interests to process large-scale graphs efficiently in science, engineering and commercial applications. Many real-world problems, such as online social networks, web graphs, user-item matrices can be represented as graph computing problems[1−4].

A real-world graph is often too large to fit in the memory of a single machine. Out-of-core graph processing is an area of technology that enables the processing of such graphs on a single machine by using disks efficiently. Disk I/O becomes the bottleneck, and many techniques have been developed to ameliorate the I/O burden[5−12].

This paper presents a new system called Cacheap for managing memory in out-of-core graph processing. Cacheap is a generic memory layer between the disk and a graph processing system or synonymously, a graph engine. Using Cacheap, a graph processing system accesses data in memory and does not perform I/O directly.

Cacheap solves three problems of memory management. The first is memory allocation, which supports variable size data and uses as much memory as possible to maximize the rate of graph processing. The second is memory caching, which uses as much memory as possible for cache and maximizes data reuse in cache. The third is memory utilization, which maximizes the portion of memory that stores graph data. Naturally graph processing and caching compete for memory, and the total memory they could use depends on memory utilization. Hence, all the three problems are tightly

Regular Paper

This work is supported by the National Key Research and Development Program of China under Grant No. 2017YFB1003103, the National Natural Science Foundation of China under Grant Nos. 61432018, 61432016, 61332009, and 61521092, the National Science Foundation of USA under Contract Nos. CCF-1717877 and CCF-1629376, and an IBM CAS Faculty Fellowship.

*Corresponding Author

©2019 Springer Science + Business Media, LLC & Science Press, China

inter-related and require coordinated solutions.

Cacheap solves the three problems together. It divides the memory between a heap and a cache. The heap manages the data used in on-going graph processing, and the cache manages the remaining data for later reuse. Cacheap uses the heap and the cache together and hence is named by the amalgamation of the two words. It uses three techniques: collaborative graph cache, variable granularity memory management, and computation-I/O overlapping. By controlling the heap and the cache together using these three techniques, Cacheap jointly maximizes parallelism, reuse and utilization.

Without a generic layer like Cacheap, existing graph engines have to use custom designs. GraphChi[13] and X-stream[9] do not use cache, TurboGraph[7] and VENUS[5] use a custom LRU cache, and GridGraph[12], NXGraph[6], and MOSAIC[8] use the page cache. Custom caches complicate system design and require careful effort to optimize the system performance. Page cache is simple to use but has a fixed granularity and replacement policy that cannot be changed with the application, even when there is algorithmic knowledge about data access. In comparison, Cacheap is optimized and portable. Please note that Cacheap only manages memory; therefore it does not improve the graph representation, and it does not change scheduling.

The paper makes the following contributions:

• a new memory system called Cacheap to maximize parallelism, reuse and utilization for graph engines (Subsection 3.1), based on a new design that combines a heap and a cache (Subsection 3.2);

• three techniques, collaborative graph cache, variable granularity memory management, and computation-I/O overlapping, to manage memory together (Subsection 3.3–Subsection 3.5);

• an easy-to-use programming interface for use by graph engines, and a demonstration system using GridGraph (Section 4);

• an evaluation using three real-world graphs and four algorithms to show integer factor speedups over three state-of-the-art solutions (Section 5).

2 Motivation

In this section, we first analyze the memory management of major out-of-core graph engines. Then, we talk about the execution model of the engines, which affects the memory usage pattern. Lastly, we motivate Cacheap with an example out-of-core graph engine, GridGraph[12], which is representative and adopts many common designs in out-of-core graph engines.

2.1 Memory Management in Out-of-Core Graph Engines

In general, out-of-core graph engines divide a graph into a set of disjoint partitions and process them iteratively. The partitioning schemes are different for different engines. Graph engines such as GraphChi and X-stream adopt vertical(horizontal) partitioning while others like GridGraph and FlashGraph[14] use 2D partitioning. No matter which partitioning scheme is adopted, the graph partitions share common properties which affect the memory management of graph engines: the partitioning granularity is coarse-grained considering the I/O efficiency and the partitions are usually variable-sized due to the skewed real-world graphs. Consequently, the size of the partitions ranges from KBs to GBs, making it hard to achieve efficient memory management.

The prototype of GraphChi uses STL (Standard Template Library) vectors to store the edge lists in the partitions and wastes time in resizing and reallocating. GraphChi then chooses to use flat arrays to store the partitions and allocate memory from OS dynamically. However, the memory management is still not satisfactory due to the allocation overhead. Many following graph engines such as Galois[15] and FlashGraph propose to use hugepage to ameliorate the overhead. However, it incurs memory fragmentation.

Other graph engines like X-stream and GridGraph use statically sized and statically allocated memory buffers to store the partitions. The buffer size is usually in MBs and stays fixed. Since the size of a partition is variable, the buffered I/O also has the problem of memory utilization. There are also other problems such as single-partition computation-I/O overlapping and double copy. More discussion of buffered I/O is in Subsection 3.6.

2.2 Memory Usage Pattern

The programming models in graph engines usually fall into two categories — vertex-centric (e.g., GraphChi and FlashGraph) and edge-centric (e.g., X-stream and GridGraph). No matter which programming model is used, the execution of most of the graph engines is the same. The partitions are periodically computed among iterations. In each iteration, the

partitions are loaded and computed in a fixed order. For some graph algorithms such as BFS (Breadth-First Search) and Connected Components, not all the partitions are needed in every iteration. Most of the graph engines support selective scheduling to skip the unused partitions. The rest partitions still follow the fixed order.

Next, we introduce a graph engine, GridGraph, in particular to better investigate the memory management problem in out-of-core graph engines.

2.3 GridGraph[12]

GridGraph is a recent out-of-core graph processing system. It is representative and adopts many common designs in out-of-core graph engines. In particular, GridGraph uses a 2D grid graph representation similar to the representation used by a variety of existing systems[8,9,13,16−20].

In the 2D grid representation, vertices are divided into P chunks and the edges are grouped into $P \times P$ partitions. Partition (i, j) contains all the edges whose source vertex is in chunk i and destination vertex in chunk j. Fig.1(a) shows an example graph with four vertices, and Fig.1(b) shows its grid representation for $P = 2$.

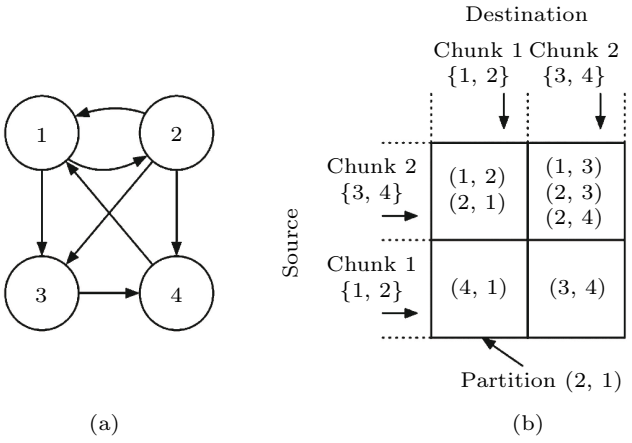


Fig.1. Graph representation in GridGraph. (a) Example graph. (b) Grid representation.

For real-world graphs, the sizes of the partitions follow a power-law distribution with a large range. For example, a graph released by Twitter has 42 million nodes and 1.5 billion edges[21]. When represented in the 32×32 grid, the smallest partition is 8.1 KB, while the largest partition is 2.3 GB.

With the grid representation, GridGraph employs an edge-centric programming model. The system iterates over the partitions in the gird either row by row or column by column. For each partition, it streams over the edges in the partition and applies a user-defined function which may make updates to the vertices in the corresponding chunks.

For iterative graph applications, the edge streaming process repeats in each iteration. GridGraph uses selective scheduling to skip reading inactive partitions which are not used in an iteration. The selective scheduling is also adopted by a variety of existing systems[8,9,13,17,19,22]. GridGraph implements selective scheduling by maintaining a vertex bitmap. Only the vertices that need to be updated in the next iteration are set in the bitmap in the current iteration. We call these vertices with the bits set active vertices. A partition is active if any of the vertices in its corresponding source chunk is active; otherwise, it is inactive. With the bitmap, GridGraph can skip loading and computing the inactive partitions.

In out-of-core processing, most of the execution time may be consumed by I/O, i.e., reading graph data from disk and writing it back. As a demonstration, we have tested GridGraph for two real-world graph datasets, for a total of eight tests, on a machine with 8 GB memory. The two graphs are 11 GB (Twitter)[21] and 28 GB (UK, a real-world web graph)[23], respectively, which are both larger than the memory size (8 GB).

Fig.2(a) shows the decomposition of the total execution into computation and I/O. Across the eight tests, the percentage of execution time for disk I/O ranges from 77.2% to 99.3%. The result is consistent with previous findings in other out-of-core graph processing systems[8,10]: I/O is the bottleneck in all tests.

The I/O bottleneck is caused by limited memory. Fig.2(b) shows the amount of I/O as a factor of the size of the input graphs, 11 GB for Twitter and 28 GB for UK. Across the eight tests, the amount of I/O is 7.9 times–37 times of the graph size, demonstrating that most of the disk accesses happen because the graph data previously loaded into memory has been evicted before the next access. There is a great potential that most of the I/O may be made unnecessary by good memory management.

There are two questions for I/O optimization: what the minimal amount of the unavoidable I/O with limited memory is, and how much a practical system can be implemented to achieve this lower bound in I/O. The graph engines often have knowledge about the data access. For example, because of the selective scheduling in GridGraph, the partitions to be used in the next iteration are known in the current iteration. This raises

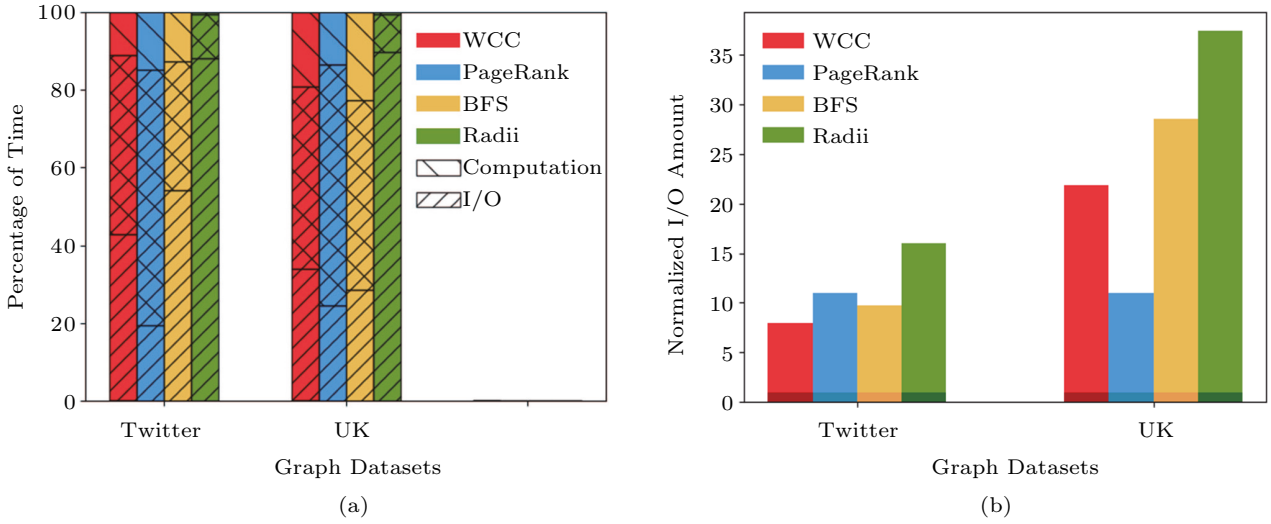


Fig.2. Performance analysis of GridGraph running four algorithms on two graphs. (a) I/O takes most of the execution time. (b) Most I/O is for loading graph data repeatedly.

the possibility of application-directed memory management.

3 Cacheap Design

In this section, we first motivate the design of Cacheap and then introduce its components and organization. Then we describe the three techniques adopted — collaborative graph caching, variable granularity memory management, and computation-I/O overlapping — and their benefits.

3.1 Design Goals

Cacheap is a generic memory layer between a graph engine and a file system. Using this memory layer, a graph engine performs all graph processing by accessing data in the memory and does not perform I/O directly. On behalf of the graph engine, Cacheap handles all data access at the file system by reading the graph data from the disk into the main memory and writing the modified data from the main memory back to the disk.

Cacheap is so named because it consists of mainly two components, a heap and a cache. The two components are needed for different purposes.

• *Heap.* Based on the available memory and the speed of I/O, the heap stores the maximal amount of active data for graph processing.

• *Cache.* All additional memory not used by the heap is used as a cache, which supports automatic and application-directed cache management.

Cacheap is designed to provide the highest memory performance for the host graph engine. In particular, it is to achieve the following four goals.

• *Maximal Parallelism in Graph Processing.* The heap enables the maximal number of threads allowed by the limited memory size and the slow speed of disk I/O.

• *Minimal I/O.* The cache enables the maximal data reuse allowed by the limited memory, which minimizes repeated data access in the disk.

• *Maximal Memory Utilization.* The heap and the cache are managed together to provide near 100% memory utilization, i.e., the portion of physical memory that stores graph data.

• *Complete Computing and I/O Overlapping.* The heap and the cache operate together so that when I/O is the bottleneck, I/O time is always overlapped with computing, and when computing is the bottleneck, computing is always overlapped with I/O.

The four goals of Cacheap have conjunctive relations. The combination of the second and the third goals means that Cacheap stores the most graph data in the memory and makes the most reuse when the data is in the memory. The combination of the first and the last means that when computing is the bottleneck, the computing time is never interrupted by I/O, and the computing always runs forward at the greatest parallelism. With the joint control, Cacheap supports the highest memory performance.

Previous work manages memory separately and may achieve the first goal but not the next three goals. We

will discuss the differences between the joint and the separate memory control in Subsection 3.6.

3.2 System Architecture

The high-level organization of Cacheap is shown in Fig.3. It is a memory layer between the graph engine and the OS file system. It divides the memory into the heap and the cache. The system interacts asynchronously through a set of queues. Fig.3 shows two queues, a request queue and a ready queue, which are used to communicate between Cacheap and the graph engine.

From the perspective of Cacheap, the graph engine consists of a set of computing threads. Before processing a graph partition, the graph engine inserts a request for the partition into the request queue. Cacheap serves the request from the request queue and loads the graph partition into the heap. It then sends a ready signal to the ready queue, which activates the processing by the graph engine.

We use the term heap because it supports the allocation and deallocation of variable-size graph data, and it has a bounded capacity. We refer to the data in the heap as active data. It includes all data currently being processed by the graph engine and the data being loaded by the current request. The rest of the data in memory is inactive. The limited memory bounds the heap capacity, which bounds the maximal amount of active data and in turn bounds the amount of parallelism.

Effectively, the heap implements memory-bound parallelism control. It is analogous to a task queue, which is CPU-bound parallelism control. Just as a task queue is used to run as many tasks as the number of available cores, the heap in Cacheap is used to run as many threads as the amount of memory permits.

The heap and the two queues maximize the parallelism in graph processing. With sufficient threads in the graph engine, the ready queue is always empty, and graph processing starts immediately after a graph partition is loaded into the heap. When the memory is lim-

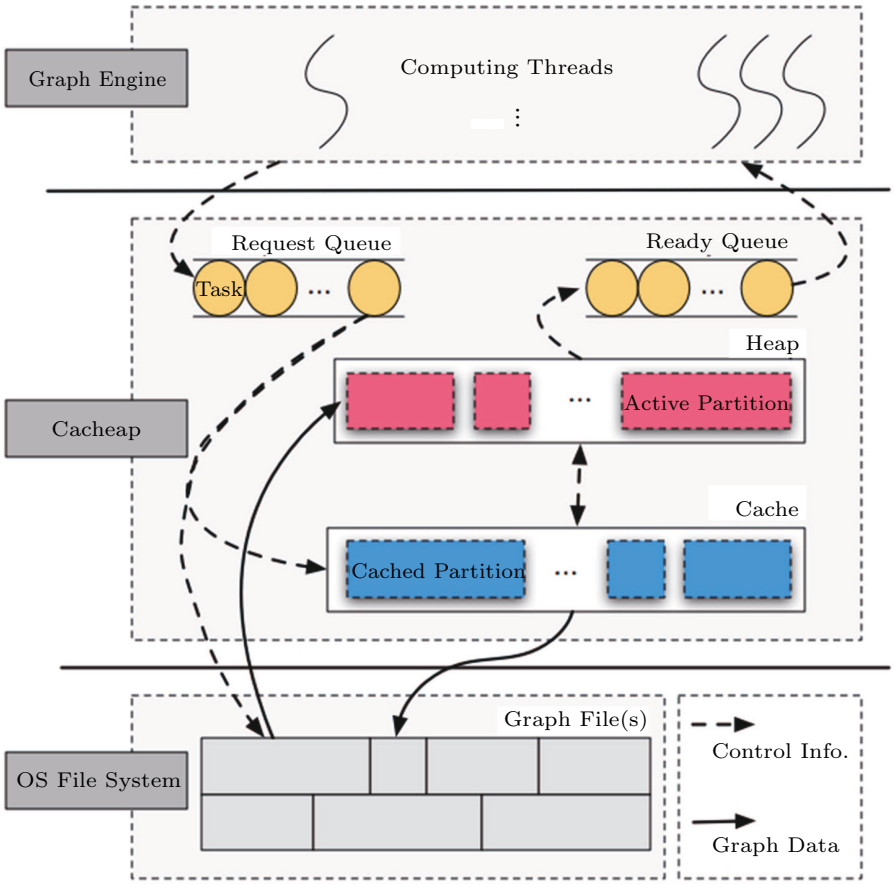


Fig.3. High-level design of Cacheap: it is a generic memory layer between the graph engine and the file system, and it divides the memory into a heap and a cache.

ited and the disk is fast, the request is filled as quickly as the data is computed, and all the available memory will be given to the heap. When the disk is slow, however, the request is filled slower than the data is computed, and the amount of active data, i.e., the heap size, may be smaller than the memory size.

Hence, if I/O is sufficiently slow, there will be memory not needed by the heap, and this remaining memory is given to the cache.

The cache stores the inactive data which is currently unused but may be used again in the future. As a generic system, the cache in Cacheap can support any type of cache replacement. We have implemented LRU and MRU polices. More importantly, we have developed collaborative graph caching (CGC), which we describe in Subsection 3.3. Any graph engine using Cacheap can choose any of its caching policies. In evaluation, we will show that the new CGC policy closely approximates optimal performance.

The heap and the cache work together to serve the graph engine. When the next request in the request queue is served, the graph partition or part of it may be stored in the cache, and it will be moved directly to the heap. The missed data will be loaded from the disk. The process involves finding free memory and scheduling the disk transfer, which we will describe in Subsection 3.5.

Cacheap supports variable granularity data, as indicated by different size blocks in Fig.3. Such support is necessary, because the unit of data in graph processing may vary, for example, by several orders of magnitude in GridGraph as discussed in Section 2. Cacheap stores each partition in consecutive blocks of a fixed size. It manages the data in the heap and the cache together. The joint management is necessary for memory utilization and the other goals mentioned in Subsection 3.1. In particular, the design enables fine-grained (block-granularity) computation-I/O overlapping, which we explain in Subsection 3.6, and near 100% utilization of memory, which we will show in evaluation.

3.3 Collaborative Caching

We first describe the basic cache operations and then a policy called collaborative graph cache (CGC).

3.3.1 *Cache Operations*

The cache is managed by a cache manager, which maintains a hash table for cache lookup and a set of cache replacement policies. When a graph partition is finished, its meta data is inserted into a queue called the release queue. The cache manager dequeues the release and moves the graph partition from the heap to the cache. Note that the move is logical and entails adding the meta data to the hash table. There is no physical copying of the graph partition.

When memory is needed for loading a request, the cache manager uses the cache policy to select the graph partition to evict. If the requested data is in the cache, i.e., a cache hit, the data is (logically) moved to the heap. In both cases, evicting to the disk and moving to the heap, the meta data is deleted in the hash table. The eviction may have the extra operation of writing back the data if it is modified.

3.3.2 *CGC Policy*

Most of the graph algorithms such as Breadth-First Search (BFS) and Connected Components (CC) are iterative. In each iteration, a subset of the graph needs to be processed. The subsets in successive iterations may differ. However, it is often the case that the subset used in the next iteration is determined in the current iteration. Based on this observation, we propose CGC as a collaborative cache policy to exploit the application knowledge for iterative graph processing.

Through the Cacheap programming interface (see Subsection 4.1), a graph engine can signal Cacheap that a graph partition will be needed in the future. We call the signal a hint and the graph partition the hinted data.

CGC records hints in a set. It favors eviction of unhinted data in the cache. This is implemented by a stack shown in Fig.4, where the hinted data is grouped near the top and the unhinted near the bottom.

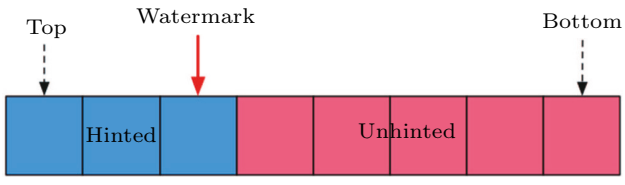


Fig.4. Joint management of hinted and unhinted data in CGC.

Logically, the stack of CGC consists of two stacks, hinted and unhinted, with a dynamic boundary. The replacement in each stack can use any policy. In CGC, we use the most recently used (MRU) policy. When the hinted data is added to the cache, or the unhinted data becomes hinted, it is added to the bottom of the hinted section. Eviction always happens at the bottom of the combined stack.

MRU is used in CGC on the assumption that the order a graph engine traverses graph partitions is the same across iterations. For example, the order can be row-by-row or Hilbert curve for a 2D grid representation (Section 2). MRU is the most effective because the most recently used partition is likely accessed the furthest in the next iteration, if it is used.

3.3.3 Comparison with MRU and OPT

Because it combines cache hints and MRU, CGC outperforms MRU. In particular, CGC never evicts partitions to be used in the next iteration as long as the cache can hold all the hinted partitions, while MRU may evict these partitions. This is shown by the example in Fig.5. Suppose the partitions are equal-sized, and the cache size is 3. At time 3, a, b, c are in cache and we need to evict one to make room for d. For CGC, supposing c is hinted, CGC will choose to evict b because b is the most recently used in the unhinted set. The two accesses in the second iteration will both be hits. However, MRU would evict c and incur a miss at time 4.

Partition Trace	$a\ b\ c\ d$	$c\ d$	$b\ d$
Logical Time	0 1 2 3	4 5	6 7

Fig.5. Example sequence of eight accesses to four graph partitions. The accesses happen in three iterations, separated by dotted lines.

Belady gave the optimal off-line policy, which evicts the block that is accessed the furthest in the future when a replacement happens[24]. Mattson *et al.* showed how it can be implemented for all cache sizes and called it the OPT algorithm[25]. CGC is not optimal. However, there are only two scenarios that CGC performs worse than OPT.

• The victim is not used in the next iteration but used in at least one of the iterations after the next iteration (suppose the next iteration is i and the earliest iteration in which the victim is used is j) and no prior partition is used between the iterations i and j.

• The victim is used in the next iteration but has not been hinted before the eviction.

For the example in Fig.5, the victim b at time 3 is not used in the second iteration but used in the third iteration. The eviction of b will lead to a miss at time 6. However, the miss can be avoided in OPT because OPT will evict a instead of b because a is not used between the second and the third iteration. For CGC, supposing c has not been hinted at time 3, CGC treats a, c the same and may evict c, causing a miss at time 4.

However, in real-world graph algorithms, either of the two scenarios seldom happens. The first scenario has very strict constraint conditions. For the second, a partition is usually immediately hinted if it is to be used in the next iteration. Evaluation in Subsection 5.5 shows that the performance of CGC is very close to that of OPT in real-world graph analytics.

3.4 Variable Granularity Memory Management

As a memory layer, Cacheap manages data objects of any size by allocating consecutive memory in the address space. In a graph engine, although the size of the graph partitions varies, it is usually much larger than the size of a physical memory page. Based on this observation, Cacheap manages them in blocks of a constant size that is multiples of the page size.

Cacheap keeps track of its memory in blocks. The size of maximal physical memory Cacheap can use is a user-defined parameter and is set during Cacheap initialization. Physical memory is consumed by the graph partitions stored in the memory, regardless of whether they are in the heap or in the cache. In fact, both the heap and the cache are mere logical constructs. There is never data copying when a graph partition is moved between the heap and the cache.

A graph partition may be partially resident in the memory, that is, some but not all of its blocks are stored in memory, and the remaining blocks have to be loaded from the disk. Cacheap uses the system call *mmap* to allocate memory to load a new graph partition, *mremap* to allocate memory to complete loading of a partially resident partition, and *munmap* to deallocate the memory used by a graph partition. To observe the memory bound, Cacheap calls the allocation *mmap* only after sufficient deallocation by *munmap*.

Cacheap evicts cache data at the block granularity. To satisfy a memory allocation, it may suffice to deallocate just a part of a graph partition in the cache. A consequence is a partial cache hit, where some but not all of the blocks of the requested graph partition are in the cache. The cache returns the partially cached data immediately to the heap and at the same time loads the missing blocks from the disk. To enable graph processing on partial partition, the cache evicts the last block of a graph partition first, and its subsequent eviction

moves backwards. Depending on the design, a graph engine may start computing on the beginning blocks of the partial graph partition, while the loading of the missing later blocks happens in parallel.

In CGC, the hints are given dynamically. It is possible that a hint comes when a graph partition is in the middle of eviction. Since the eviction happens block by block, the not-yet-evicted blocks are rescued and moved to the hinted section of the cache. This is another reason for a partial cache hit.

3.5 Computation and I/O Overlapping

The request processing is asynchronous in Cacheap, so is I/O processing. The two methods are shown in Algorithm 1. When the request queue is not empty, the next request is processed by the method *ProcessRequest*. It first tries to serve the requested data from the cache. If the data needs to be loaded from the disk, the system will check whether there is enough free memory available for the heap to store it. In most cases, there is no free memory, and the cache will evict the right amount of cached data to free just enough memory to give to the heap. The eviction happens in block granularity as discussed in Subsection 3.4. After eviction, it schedules I/O by inserting a disk-loading request to the I/O queue. Note that evicted data still resides in memory after eviction and remains so until the I/O request is served.

Algorithm 1. Request & I/O Processing Methods

```
1: function ProcessRequest()
2:     Key = Pop(ReqQueue)
3:     P = CacheLookup(Key)
4:     Mark P as active
5:     if P in cache then
6:         CacheDelete(Key)
7:         Push(ReadyQueue, P)
8:     else if part of P in cache then
9:         CacheDelete(Key)
10:        Push(ReadyQueue, P1)          ▷ cached P1
11:        Push(IOQueue, P2)             ▷ uncached P2
12:    else                              ▷ P not in cache
13:        Push(IOQueue, P)
14:    end if
15: end function
16: function ProcessIO()
17:    P = Pop(IOQueue)
18:    if no enough free memory then
19:        CacheEvict(P.size)
20:    end if
21:    Read(P)                           ▷ Read from disk
22:    Push(ReadyQueue, P)
23: end function
```

When the request queue is not empty, the next request is processed by the method *ProcessIO* shown in Algorithm 1. It first deallocates the evicted data, by *unmap*, and then allocates the memory for the requested data, by *mmap* for a whole graph partition or *mremap* for a partial partition. After I/O is completed, it pushes the meta-data to the ready queue. Cacheap supports parallel I/O, where the I/O queue can be served by multiple disks.

In Cacheap, the moving of data from the heap to the cache uses one more queue, the release queue. After finishing computing on a graph partition, the thread will add the partition to the release queue. When the release queue is not empty, the cache processes the next release. If a partition is not used by any other thread, it will be marked as inactive and moved to the cache. Reference counting may be used for heap management.

We can now analyze the computation and I/O overlapping in three cases. First, when I/O is the bottleneck, we show that I/O is always overlapped with computing. In this case, I/O queue is always filled. Hence, I/O operation is continuous. The ready queue is always emptied immediately when a graph partition is loaded, because there are always computing threads waiting. Since a graph partition is placed in Ready Queue immediately after it is loaded, I/O is overlapped with the maximal amount of processing possible. Second, when computation is the bottleneck, e.g., when there is sufficient memory to store all graph partitions, the ready queue is always filled, and all threads are busy processing, and they are completely asynchronous from I/O. Third, when computation and I/O are balanced, the I/O queue and the ready queue are both filled, and both computation and I/O are continuous at the peak speed.

3.6 Comparison with Buffer + Page Cache

Some graph engines such as X-stream and GridGraph use application buffers for graph processing and the system page cache for file caching (see Section 6). In these systems, the heap and the cache are managed separately. The joint management by Cacheap has the following benefits.

Memory Utilization. A graph engine such as GridGraph allocates a buffer of a fixed size for each thread. If a graph partition is too large, it is broken into "batches" of the buffer size. When a buffer is only partially utilized, the unused memory is the internal fragmentation, which can be large for big buffers, e.g.,

24 MB in GridGraph. Since Cacheap manages memory in blocks, and a block is always used, by either the heap or the cache, Cacheap loses utilization only for internal fragmentation in its blocks (whose size is 4 KB as explained in Subsection 5.6).

Partition-Aware Eviction. When buffers are used, a thread cannot start computing, until it finishes loading a graph partition (or a batch). The thread is blocked for I/O even though there may be partition data cached in the page cache. The joint control in Cacheap is partition-aware in that it evicts one graph partition at a time. In comparison, a page cache is partition-unaware and may evict pages indiscriminately from potentially a large number of partitions.

Singe-Partition Computation-I/O Overlapping. Cacheap may evict only a part of a graph partition. As explained in Subsection 3.4, it may have a partial cache hit, and for that partition, the computation (on the partially cached data) and the I/O (for the missed data) happen in parallel. This is impossible without the combined control. The benefit is highly prominent for graph processing, because the power-law distribution of graph edges means that there always are some partitions that are exceedingly large, e.g., 2.3 GB mentioned in Section 2.

Delayed Eviction. Another important benefit for memory utilization comes from I/O Queue. As shown in Algorithm 1, deallocation due to cache eviction is delayed while the disk request is waiting in I/O Queue. This delay maximizes the time that the evicted data resides in the memory. A related benefit comes from block granularity memory management. The combined effect is that only the minimal amount of in-memory data is evicted, and it is evicted at the latest possible time.

Single Copy. Buffered I/O involves double copy, i.e., it needs to copy data to page cache and further to user buffer. Since Cacheap operates the heap and the cache together with memory blocks, it only needs to copy the data once from disks to the memory blocks. There is no data copy between the heap and the cache. It only needs to change a bit indicating whether a partition is in the heap or the cache.

4 Cacheap Usage

Cacheap may be used by any type of graph engines. We first show its programming interface and then an example integration with GridGraph.

4.1 Cacheap Programming Interface

Cacheap has two programming interfaces. The first provides memory management, which is the heap interface. The second provides cache management, which is optional and may be used for either specific support of a graph engine or generic extension of Cacheap.

Memory Interfaces. A graph engine requests and uses memory using Cacheap through three interface functions: *Request*, *Get* and *Release*. Their parameters are given in Algorithm 2. When calling *Request*, the graph engine provides a key, which contains the file location or a file-mapping function; therefore Cacheap knows where to retrieve the data. Naturally, the key is unique for each graph partition.

Algorithm 2 . Cacheap Interfaces

```
function Require(Key)
    Push(ReqQueue, Key)
end function
function Get()
    if ReadyQueue empty then
        Wait(RQMutex)                    ▷ sleep until ready
    end if
    partition = Pop(ReadyQueue)
    return partition
end function
function Release(Partition)
    Push(RelsQueue, Partition)
end function
function Hint(Args)
    Push(HintQueue, Args)
end function
```

Each of *Request* and *Release* has a single parameter and registers a need and a release for memory respectively. They are non-blocking. *Get* has no parameter and takes the next graph partition for processing. It is blocking. Their implementation is shown in Algorithm 2. *Request* and *Release* insert requests into the request queue and the release queue respectively. *Get* retrieves from the ready queue. The request queue and ready queue are illustrated in Subsection 3.2, and the release queue is described in Subsection 3.5.

The memory interface of Cacheap is simplified because of its support of data objects of any size. It is well known that in real-world graphs, the size of graph partitions is not uniform and follows the power law distribution[26]. In the past, a graph engine must handle variable-size partitions. This adds extra burden to system complexity and can impose unnecessary limits. For example, GridGraph allocates a fixed-size buffer for each computing thread. If the thread loads a partition

that is larger than the buffer size, the thread has to divide the partition into pieces (batches) and load them separately. However, with Cacheap, graph engine programmers do not need to care about the management of memory for the variable-size partitions (e.g., buffers in GridGraph)

Cache Interface. Cacheap provides an interface optionally used for implementing customized cache policies. A graph engine can define a new cache policy class by inheriting from the abstract class given in Fig.6, with three abstract functions, *add*, *evict* and *delete*. They must be implemented in the new cache policy class. A fourth function, *hint*, can be used if the new policy is collaborative as CGC.

```
class CachePolicy {
 virtual void add(Partition) = 0;
 virtual void delete(Partition) = 0;
 virtual Partition* evict(size_t) = 0;
 virtual void hint(int n, ...){}
};
```

Fig.6. Abstract CachePolicy class.

Cacheap calls the *add* function when moving a partition from the heap to the cache and *delete* when moving a partition in the opposite direction from the cache to the heap. Both functions take the moving partition as the parameter. When there is not enough free memory to load a partition, Cacheap calls the *evict* function to decide what to evict. The size of needed memory is the parameter to the *evict* function. The function then returns a list of victim partitions selected by the new policy.

It takes a careful design to manage memory with good utilization and cache with high efficiency. A custom design of the memory layer is either inferior or a repetition of effort, if it is fully effective. By using Cacheap instead, a graph engine can avoid the complexity of memory management entirely and yet benefit from maximizing parallelism, minimizing I/O, and fully overlapping computation with I/O.

4.2 Integration in Graph Engines

In this subsection, we show how graph engines use the Cacheap interfaces with GridGraph as an example. GridGraph adopts the edge-centric programming model and streams over the partitions containing edges. The total modification is light, about 60 lines. In particular, we modify GridGraph as follows.

1) The main thread calls the *Require* interface for each active partition. We use the coordinate of the partition as its key.

2) The computing threads retrieve ready partitions by calling the *Get* interface, process the edges in the partitions and release them when finished.

The process is listed in Algorithm 3. The function F is a filter function to skip computing the inactive vertices. F_e is a user-defined function for edge processing.

Note that each partition associates with a key. For GridGraph, the key is an integer pair, (a, b), indicating the partition coordinate in the grid.

We also implement CGC in GridGraph by integrating the CGC hints in its selective scheduling. GridGraph uses a bitmap to indicate the activeness of vertices. It provides an interface function for graph application programmers to mark a vertex as active and to be used in the next iteration. We extend the function to find the corresponding active partitions and pass them to Cacheap by calling the *Hint* interface.

Algorithm 3. GridGraph Using Cacheap

```
function StreamEdges(F_e, F)
    fork Compute(F_e, F)
    for each active partition① P do
        Require(P.key)
    end for
end function
function Compute(F_e, F)
    Sum = 0
    while not all active partitions done do
        P = Get()
        for each edge in P do
            if F(edge.source) then
                Sum += F_e(edge)
            end if
        end for
        Release(P)
    end while
    return Sum
end function
```

5 Evaluation

We evaluate Cacheap in this section. Cacheap is designed to optimize I/O for out-of-core graph engines. To demonstrate Cacheap, we adopt it in GridGraph, which is a representative out-of-core graph engine. To show its portability, we also extend Cacheap to other two graph engines, GraphChi and X-stream.

①In GridGraph, an active partition is a partition with edges to be processed.

5.1 Experimental Setup

In our evaluation, we use four graph algorithms, WCC, Radii, BFS and PageRank, which are all implemented in GridGraph. WCC stands for weakly connected components. It finds subsets of vertices connected with edges by propagating the subset labels via the edges iteration by iteration until convergence. BFS is breadth-first search. In the first iteration, only the chosen root vertex is active and all its unvisited neighborhood will be marked as active in the next iteration. The process ends until no vertex is active. Radii finds the radius of a graph. It starts from 64 random selected vertices doing breadth-first search and updating the radius for each vertex until all the radii are found. PageRank approximates the impact of each vertex on the graph by propagating the impacts over the edges. We run PageRank for 10 iterations in our experiments.

Four real-world graphs, listed in Table 1, are used as inputs in our evaluation. They are all billion-edge graphs with highly skewed power-law degree distribution.

Table 1. Graph Datasets

Graph	$\|V\|$ ($\times 10^6$)	$\|E\|$ ($\times 10^9$)	Size (GB)	Type
Twitter	42	1.5	11	Social
UK	106	3.7	30	Web
Yahoo	1 400	6.6	53	Web
Clueweb	978	42.6	336	Web

Note: $|V|$ means the number of vertices and $|E|$ means the number of edges.

We conduct all the experiments on AWS EC2 i2.2xlarge instances. We choose to use the i2.2xlarge instance because it is storage optimized providing direct-attached SSD with high I/O bandwidth. The instances are storage optimized and recommended by AWS to run applications that can benefit from high I/O performance. The i2.2xlarge instance has eight hyperthread vCPU cores, 61 GB memory and 2×800 GB SSD storage capacity. We run eight threads in our tests. The actual I/O bandwidth is about 470 MB/s tested with *fio*.

We modify GridGraph, GraphChi, and X-stream lightly (tens of lines, including CGC hints). We do not modify any of the four graph applications used in our evaluation.

The baseline GridGraph adopts a heuristic mechanism to determine whether to use OS page cache: page cache is used only if the total size of all the partitions used in an iteration does not exceed the memory. In order to make a better comparison of page cache and Cacheap, we make another modification of GridGraph, in which page cache is always used. We call this version PageCache.

5.2 Overall Performance

We report the performance of GridGraph and its two modifications PageCache and Cacheap in Table 2, for different memory sizes.

When the memory size is smaller than the graph size, Cacheap outperforms GridGraph in all tests with speedup ranges from 1.29x to 3.86x for WCC, 1.26x to 2.38x for PageRank, 1.27x to 4.49x for BFS, and 1.13x to 1.98x for Radii. On average across all algorithms, the improvement ranges from 1.30x for 11 GB Twitter graph at 4 GB memory to 3.43x for 30 GB UK at 24 GB memory. On average across all tests, Cacheap is twice (2.04) as fast as GridGraph.

When the memory size is the same with or larger than the graph size, caching does not matter. Cacheap still outperforms GridGraph in all tests except for two, in which it is slower by a negligible amount. We will evaluate the overhead of Cacheap shortly. On average, Cacheap is 1.15x as fast as GridGraph, because Cacheap overlaps computation and I/O completely.

When the memory size is smaller than the graph size, GridGraph uses I/O directly without caching. A second solution, PageCache, uses the file cache, but the performance is very similar to that of GridGraph, showing little effect by the page cache.

Cacheap outperforms PageCache as significantly as it does GridGraph. The improvements of Cacheap over these two alternatives come from the benefits discussed in Subsection 3.6 and from better caching to be shown in Subsection 5.5.

We also show the Cacheap speedups over GraphChi and X-stream in Table 3. Similar to the results on GridGraph, Cacheap consistently outperforms GraphChi and X-stream. On average across all tests with memory smaller than the graph size, Cacheap is 3.48x and 2.78x faster than GraphChi and X-stream, respectively.

5.3 Computation and I/O Overlapping

Fig.7 shows the computation time and the I/O time for the first nine iterations of WCC computing on the Twitter graph.

Table 2. Execution Time (in Seconds) of Four Graph Algorithms on Four Graphs with GridGraph, PageCache and Cacheap and the Speedup of Cacheap over GridGraph under Different Memory Sizes

Graph Algorithm	System	Twitter (11 GB)			UK (30 GB)			Yahoo (53 GB)			Clue (336 GB)
		4 GB	8 GB	12 GB	16 GB	24 GB	32 GB	32 GB	48 GB	61 GB	61 GB
WCC	GridGraph	187.7	180.7	37.1	1 451	1 371	205.9	12 240	7 246	1 562	79 265
	PageCache	187.1	180.1	37.42	1 433	1 357	206.9	12 127	5 458	1 554	79 032
	Cacheap	116.4	73.51	34.45	723.8	354.8	146.3	8 879	2 961	1 510	61 328
	Speedup	1.61x	2.46x	1.08x	2.00x	3.86x	1.41x	1.38x	2.45x	1.03x	1.29x
PageRank	GridGraph	269.4	270.9	96.38	703.5	695.7	232.1	3 065	2 300	2 268	24 812
	PageCache	283.8	273.3	94.11	706.3	693.4	233	2 122	1 983	1 989	24 690
	Cacheap	229.8	141.7	96.8	450.5	292.2	184.8	2 015	1 829	1 940	17 035
	Speedup	1.71x	1.91x	1.00x	1.56x	2.38x	1.26x	1.52x	1.26x	1.17x	1.46x
BFS	GridGraph	230.5	226.1	38.27	1 786	1 747	183.4	9 845	5 601	1 320	69 274
	PageCache	235.1	223	39.09	1 790	1 738	183.5	7 685	2 483	1 326	62 886
	Cacheap	181.9	105.8	36.05	893.3	398.4	129.3	5 051	1 778	1 346	30 727
	Speedup	1.27x	2.14x	1.06x	2.00x	4.39x	1.42x	1.95x	3.15x	0.98x	2.25x
Radii	GridGraph	404.5	399.2	106.1	2 826	2 468	308.7	36 274	16 406	14 654	211 255
	PageCache	426.13	406.26	106.22	2 656	1 893	313.8	36 168	16 742	11 644	211 289
	Cacheap	356.9	201.2	106.1	1 600	794.5	295.44	28 136	13 209	11 241	175 447
	Speedup	1.13x	1.98x	1.00x	1.77x	3.11x	1.04x	1.29x	1.24x	1.30x	1.20x
Average		1.30x	2.12x	1.03x	1.83x	3.43x	1.28x	1.53x	2.02x	1.12x	1.55x

Table 3. Cacheap Speedups over GraphChi and X-stream Under Different Memory Sizes

Graph Algorithm	System	Twitter (11 GB)			UK (30 GB)			Yahoo (53 GB)			Clue (336 GB)
		4 GB	8 GB	12 GB	16 GB	24 GB	32 GB	32 GB	48 GB	61 GB	61 GB
WCC	GraphChi	1.89x	3.21x	1.10x	2.12x	3.46x	1.32x	1.89x	3.22x	1.11x	1.57x
	X-stream	1.88x	2.93x	1.12x	2.45x	4.02x	1.22x	1.27x	2.88x	1.05x	1.39x
PageRank	GraphChi	2.21x	3.86x	1.26x	1.97x	4.35x	1.22x	1.81x	2.89x	1.24x	1.91x
	X-stream	1.41x	1.89x	1.04x	1.99x	3.27x	1.13x	1.86x	2.01x	1.17x	1.79x
BFS	GraphChi	1.82x	2.88x	1.32x	1.93x	4.27x	1.31x	2.04x	4.55x	1.15x	2.32x
	X-stream	1.26x	2.22x	1.01x	1.34x	2.97x	1.18x	1.92x	3.76x	1.00x	2.14x
Radii	GraphChi	1.88x	2.61x	1.05x	2.86x	5.25x	1.17x	2.53x	4.64x	1.17x	2.19x
	X-stream	1.42x	1.99x	0.99x	1.44x	1.87x	1.03x	1.43x	2.03x	1.12x	1.54x
Average	GraphChi	1.95x	3.14x	1.18x	2.22x	4.33x	1.26x	2.07x	3.83x	1.17x	2.00x
	X-stream	1.49x	2.26x	1.04x	1.81x	3.03x	1.14x	1.62x	2.67x	1.09x	1.72x

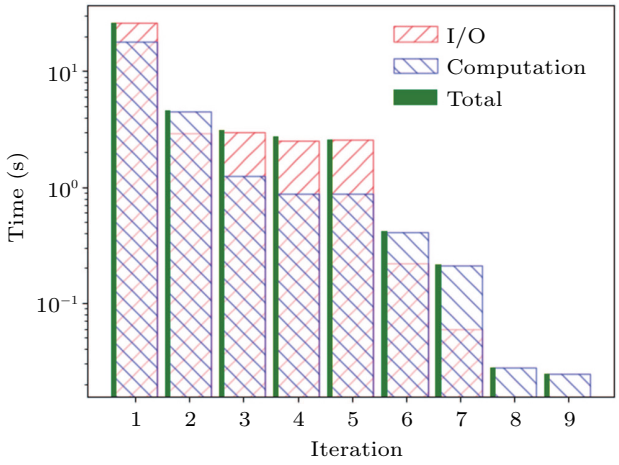


Fig.7. Computation, I/O time and total time for each iteration in WCC on Twitter with 10 GB memory.

Note that the total time taken by each iteration is always nearly the same as the computation time or the I/O time depending on which is longer. Therefore, Cacheap overlaps computation and I/O regardless of the bottleneck, which is I/O in iterations 1, 3, 4, 5 and computation in iterations 2, 6, 7, 8, 9, respectively.

For each iteration, we have measured the difference between the running time and the larger one of the computation and I/O time. This shows the degree of insufficient overlapping. For the four algorithms on Twitter with 8 GB memory and on UK with 16 GB memory, the non-overlapped time is in the range from 0.08% to 2.9% and on average 1.39% of the total run time, showing near complete computation-I/O overlapping by Cacheap.

5.4 Cacheap Overhead

Cacheap may incur overhead due to queue pop/-push, cache operations and partition allocation/deallocations. All these operations are trivial compared with the computation and I/O time for a partition.

The overhead is then calculated by dividing Cacheap time by the sum of computation time, I/O time and Cacheap time. The overhead for four algorithms on Twitter with 8 GB memory and on UK with 16 GB memory is in the range from 0.1% to 2.2% and on average 1.3%.

5.5 CGC Policy

We show the miss ratio curves of three cache replacement policies when running the four algorithms on the Twitter graph in Fig.8. For LRU, MRU and CGC, we run each algorithm for 15 times for each of the polices with the cache size ranging from 1 GB to 15 GB and the memory budget from 2 GB to 16 GB. We record the block access and miss frequency for each run, and then compute the miss ratios showed in Fig.8. Since the fourth policy, OPT, is an off-line policy which needs the access information in the future, we simulate it with a trace collected in one of the runs mentioned above.

The four algorithms have been introduced earlier. The first three algorithms, WCC, BFS and Radii, have the dynamically changing subset property described in Subsection 3.3. The subset in WCC is at first the complete graph and then shrinks as the algorithm progresses. BFS and Radii have the same pattern. The

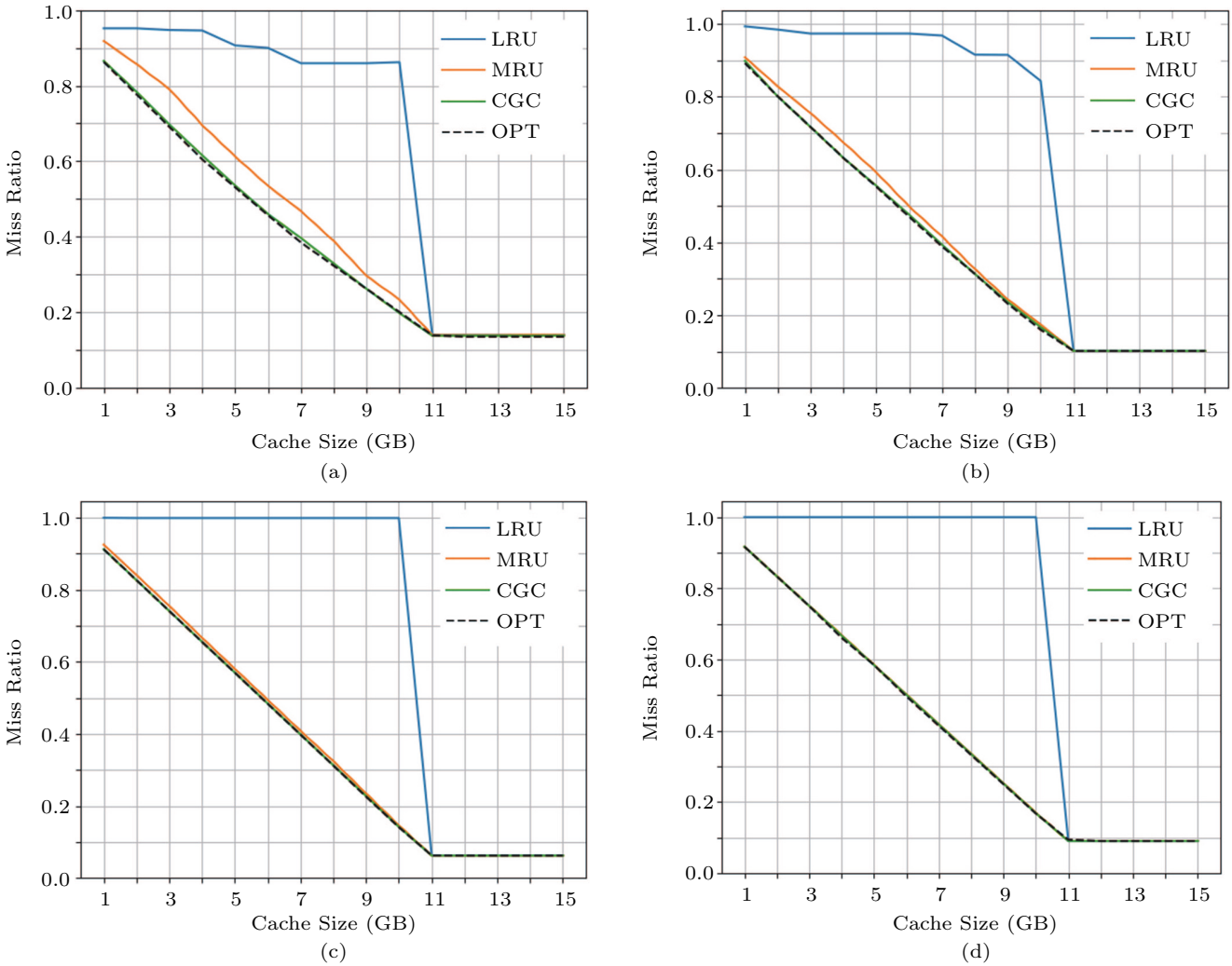


Fig.8. Comparison of the miss ratio curve of four graph algorithms computing on the Twitter graph using LRU, MRU, CGC and OPT. (a) WCC. (b) BFS. (c) Radii. (d) PageRank.

subset is small in the beginning, increases for some iterations, and then shrinks for the remaining iterations. PageRank does not have the dynamically changing subset property because it accesses the entire graph in each iteration.

CGC outperforms MRU the most in WCC and to a less degree in BFS and Radii. In PageRank, CGC and MRU perform the same. For algorithms that iterate over the entire graph in each iteration, MRU is optimal.

CGC, however, is optimal in all cases. There is no further room for improvement. This shows that the simple hint interface in Cacheap is sufficient to maximize cache performance at all memory sizes.

5.6 Memory Utilization

Cacheap manages memory in blocks. We evaluate how fully Cacheap utilizes the memory as a function of the block size. At each instant in execution, the memory utilization is the portion of the memory that is used to store graph partitions. For each run, the instant utilization is measured at each allocation, i.e., *mmap, mremap*. We report the average as the utilization for the execution.

Fig.9 shows the memory utilization with different block sizes from 4 KB to 32 MB. At small sizes such as 4 KB, the memory utilization is almost 100%. However, partial eviction happens more frequently, as indicated by the number of *mremap* calls. Such calls are expensive because they may copy data and invalidate TLB.

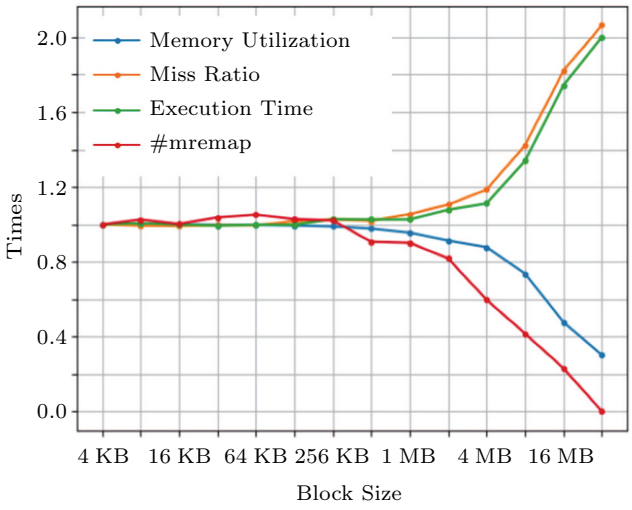


Fig.9. Memory utilization, miss ratio, execution time and number of *mremap* calls (#mremap) of WCC on Twitter using 8 GB memory with different block sizes from 4 KB to 32 MB. All the metrics except memory utilization are normalized w.r.t. the results with 4 KB block size.

We also report execution time, miss ratio and remapping counts in Fig.9. As the block size increases beyond half megabytes, the memory utilization starts to drop beyond 5%. The lower utilization reduces the usable memory and the effective cache size. As a result, the miss ratio increases, and the running time increases with the miss ratio. The relative increase of the running time is lower than that of the miss ratio. We believe that the reason is the reduction of *mremap* calls and hence their overhead.

These results show that memory utilization is highly critical to performance. It is more important than the memory allocation overhead such as remapping by *mremap*. Therefore, Cacheap uses the block size of 4 KB.

6 Related Work

Research on disk-based graph processing systems is pioneered by Kyrola *et al.* who proposed GraphChi[13]. GraphChi splits the vertices of a graph into P disjoint intervals and divides the large edge lists into shards. Each of the intervals is associated with a shard which contains all the edges whose destination is in the interval. GraphChi employs a parallel sliding window method to process the graph and the shards need to be loaded twice with $\theta(P^2)$ non-sequential disk seeks for each full iteration, making I/O take most of the running time.

Since I/O is the bottleneck of the disk-based graph processing systems, a lot of efforts[5−12,20,27,28] have been made to ameliorate the I/O burden.

TurboGraph[7] is an out-of core graph processing engine using SSDs. It proposes an adjacency list based graph representation to exploit I/O parallelism. The graph is represented by adjacency lists. The graph and the metadata for the adjacency lists are organized as pages which are of size 1 MB and consecutively stored on disk. In TurboGraph, a buffer manager is used to maintain a buffer pool which is an array of frames to store the pages in memory. Cacheap is similar to the buffer manager in TurboGraph. Both serve as memory cache for graph data. Cacheap differs from the buffer manager in granularity and portability. Cacheap supports arbitrary size data instead of fixed size frames, making it easier to use by graph engines and usable for storing all kinds of graph representations. In addition, Cacheap supports collaborative and user-defined cache replacement policies, while the buffer manager uses LRU.

VENUS[5] adopts a novel graph computing model called streamlined processing which supports the fine-grained parallelism of computation and disk I/O. The authors[5] also proposed a new graph representation with which the total amount of graph data can be reduced. We note that VENUS also exploits a memory cache. However, the cache is only used for storing vertex data, and it uses LRU as TurboGraph does.

Maass *et al.* proposed MOSAIC[8], which is a graph processing system designed for single heterogeneous machine (CPU+Xeon Phi) with fast storage (NVMe SSD). They designed a new Hilbert-order based data structure for graph representation, which can compress the graph and save up to 68.8% disk size for real-world graph datasets. They also proposed a hybrid computation and execution model, which can execute vertex-centric operations on CPU processors and execute edge-centric operations on Xeon Phi processors.

Mostly recently, Liu and Huang proposed an I/O request-centric programming model in a new graph engine Graphene[19]. We notice that, with Cacheap interface, we can easily make a graph engine become I/O request-centric. Also, the other techniques such as bitmap based adjacency list representation and workload balancing are orthogonal to Cacheap's optimizations and Cacheap is adoptable to Graphene.

Besides all these graph processing systems, Vora *et al.*[10] proposed a generic I/O optimization for disk-based graph processing. The optimization employs dynamic partitions which are dynamically adjusted partitions with only needed edges to reduce the I/O amount. There is a possibility that the needed edges are not present in the dynamic partitions and the system will fail to continue executing. The authors present a new accumulation-based programming model to solve the problem. The graph system developers need to adapt their programming model to the accumulation-based model. At the same time, there are cases where the total I/O amounts are larger than the baseline due to the reading and writing of the missing edges. Because of the dynamic partitions, the optimization takes up to 28% additional disk space. Cacheap cannot compact the graph representation as in dynamic partition, but as a generic memory layer, Cacheap does not require developers modify their programming model, guarantees to minimize the amount of I/O, and does not consume additional disk space.

7 Conclusions

In this paper, we proposed a generic I/O optimizing system for out-of-core graph engines. Cacheap adopts three techniques, variable granularity memory management, computation-I/O overlapping, and collaborative graph cache. Cacheap can be easily integrated into graph engines and take over the task of memory and I/O management. Evaluation shows that Cacheap can achieve effectively in practice full memory utilization, complete computation-I/O overlapping, and optimal graph data caching.

References

[1] Coffman T, Greenblatt S, Marcus S. Graph-based technologies for intelligence analysis. *Communications of the ACM*, 2004, 47(3): 45-47.

[2] Han W, Miao Y, Li K, Wu M, Yang F, Zhou L, Prabhakaran V, Chen W, Chen E. Chronos: A graph engine for temporal graph analysis. In *Proc. the 9th Eurosys Conference*, April 2014, Article No. 1.

[3] Jeong H, Mason P S, Barabasi A L, Oltvai N Z. Lethality and centrality in protein networks. *Nature*, 2001, 411(6833): 41-42.

[4] Xiang L, Yuan Q, Zhao S, Chen L, Zhang X, Yang Q, Sun J. Temporal recommendation on graphs via long- and short-term preference fusion. In *Proc. the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, July 2010, pp.723-732.

[5] Cheng J, Liu Q, Li Z, Fan W, Lui C S J, He C. VENUS: Vertex-centric streamlined graph computation on a single PC. In *Proc. the 31st IEEE International Conference on Data Engineering*, April 2015, pp.1131-1142.

[6] Chi Y, Dai G, Wang Y, Sun G, Li G, Yang H. NXgraph: An efficient graph processing system on a single machine. In *Proc. the 32nd IEEE International Conference on Data Engineering*, May 2016, pp.409-420.

[7] Han W, Lee S, Park K, Lee J, Kim M, Kim J, Yu H. TurboGraph: A fast parallel graph engine handling billion-scale graphs in a single PC. In *Proc. the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining,* August 2013, pp.77-85.

[8] Maass S, Min C, Kashyap S, Kang W, Kumar M, Kim T. Mosaic: Processing a trillion-edge graph on a single machine. In *Proc. the 12th European Conference on Computer Systems*, April 2017, pp.527-543.

[9] Roy A, Mihailovic I, Zwaenepoel W. X-Stream: Edge-centric graph processing using streaming partitions. In *Proc. the 24th ACM SIGOPS Symposium of Operating Systems Principles*, November 2013, pp.472-488.

[10] Vora K, Xu G Q, Gupta R. Load the edges you need: A generic I/O optimization for disk-based graph processing.

In *Proc. the 2016 USENIX Annual Technical Conference*, June 2016, pp.507-522.

[11] Zhang Y, Liao X, Jin H, Gu L, Tan G, Zhou B. HotGraph: Efficient asynchronous processing for real-world graphs. *IEEE Transactions on Computers*, 2017, 66(5): 799-809.

[12] Zhu X, Han W, Chen W. GridGraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning. In *Proc. the 2015 USENIX Annual Technical Conference*, July 2015, pp.375-386.

[13] Kyrola A, Blelloch E G, Guestrin C. GraphChi: Large-scale graph computation on just a PC. In *Proc. the 10th USENIX Symposium on Operating Systems Design and Implementation*, October 2012, pp.31-46.

[14] Zheng D, Mhembere D, Burns C R, Vogelstein T J, Priebe E C, Szalay S A. FlashGraph: Processing billion-node graphs on an array of commodity SSDs. In *Proc. the 13th USENIX Conference on File and Storage Technologies*, February 2015, pp.45-58.

[15] Nguyen D, Lenharth A, Pingali K. A lightweight infrastructure for graph analytics. In *Proc. the 24th ACM SIGOPS Symposium on Operating Systems Principles*, November 2013, pp.456-471.

[16] Chhugani J, Satish N, Kim C, Sewall J, Dubey P. Fast and efficient graph traversal algorithm for CPUs: Maximizing single-node efficiency. In *Proc. the 26th IEEE International Parallel and Distributed Processing Symposium*, May 2012, pp.378-389.

[17] Gonzalez E J, Low Y, Gu H, Bickson D, Guestrin C. PowerGraph: Distributed graph-parallel computation on natural graphs. In *Proc. the 10th USENIX Symposium on Operating Systems Design and Implementation*, October 2012, pp.17-30.

[18] Gonzalez E J, Xin S R, Dace A, Crankshaw D, Franklin J M, Stoica I. GraphX: Graph processing in distributed dataflow framework. In *Proc. the 11th USENIX Symposium on Operating Systems Design and Implementation*, October 2014, pp.599-613.

[19] Liu H, Huang H H. Graphene: Fine-grained IO management for graph computing. In *Proc. the 15th USENIX Conference on File and Storage Technologies*, February 2017, pp.285-300.

[20] Malexicz G, Austern H M, Bik J C A, Dehnert C J, Horn I, Leiser N, Czajkowski G. Pregel: A system for large-scale graph processing. In *Proc. the ACM SIGMOD International Conference on Management of Data*, June 2010, pp.135-146.

[21] Roy A, Bindschaedler L, Malicevic J, Zwaenepoel W. Chaos: Scale-out graph processing from secondary storage. In *Proc. the 25th Symposium on Operating Systems Principles*, October 2015, pp.410-424.

[22] Kwak H, Lee C, Park H, Moon B S. What is Twitter, a social network or a news media? In *Proc. the 19th International Conference on World Wide Web*, April 2010, pp.591-600.

[23] Boldi P, Vigna S. The WebGraph framework I: Compression techniques. In *Proc. the 13th International World Wide Web Conference*, May 2004, pp.595-602.

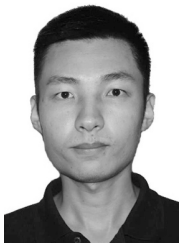
[24] Belady A L. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 1966, 5(2): 78-101.

[25] Mattson L R, Gecsei J, Slutz D, Traiger L I. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 1970, 9(2): 78-117.

[26] Faloutsos M, Faloutsos P, Faloutsos C. On power-law relationships of the Internet topology. In *Proc. the 1999 ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, August 1999, pp.251-262.

[27] Wang K, Xu H G, Su Z, Liu D Y. GraphQ: Graph query processing with abstraction refinement: Scalable and programmable analytics over very large graphs on a single PC. In *Proc. the 2015 USENIX Annual Technical Conference*, July 2015, pp.387-401.

[28] Wu M, Yang F, Xue J, Xiao W, Miao Y, Wei L, Lin H, Dai Y, Zhou L. GraM: Scaling graph computation to the trillions. In *Proc. the 6th ACM Symposium on Cloud Computing*, August 2016, pp.408-421.

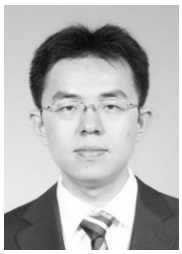


Peng Zhao received his B.S. degree in computer science and technology from Jilin University, Changchun, in 2013. Currently he is a Ph.D. candidate of Institute of Computing Technology (ICT), Chinese Academy of Sciences (CAS), Beijing. His research interests include parallel and distributed programming models and graph processing systems.

Chen Ding received his B.S. degree in computer science from Peking University, Beijing, in 1994, M.S. degree in computer science from Michigan Technological University, Houghton, in 1996, and Ph.D. degree in computer science from Rice University, Houston, in 2000. Now he is a professor and Ph.D. supervisor of University of Rochester, Rochester. Chen Ding's research seeks to understand the composite and emergent behavior in computer systems, especially its dynamic parallelism and data usage, and develop software techniques for automatic or suggestion-based locality optimization, memory management, and program parallelization.

Lei Liu received his B.S. degree in computer science from Changchun University of Science and Technology, Changchun, in 2001, M.S. degree in computer science from Jilin University, Changchun, in 2004, and Ph.D. degree in computer architecture from Institute of Computing Technology (ICT), Chinese Academy of Sciences (CAS), Beijing, in 2010. He participated in the Advanced Compiler Technology Laboratory (ACT) of ICT, CAS, in 2010, and is now an assistant professor of ICT, CAS, Beijing. His research interests include programming language and compiler optimization.

Jiping Yu is a senior student of Tsinghua University, Beijing. He is the overall winner of ASC18 Student Supercomputer Challenge, ISC18 Student Cluster Competition, and SC18 Student Cluster Competition. His research interests include high performance computing and optimization.



Wentao Han received his B.S. degree in computer science in 2008, Ph.D. degree in computer science in 2015, both from Tsinghua University, Beijing. He is currently an assistant researcher in Tsinghua University, Beijing. His research interests include design and implementation of brain-inspired software systems.

Xiao-Bing Feng received his B.E. degree in computer software from Tianjin University, Tianjin, in 1992, M.S. degree in computer software from Peking University, Beijing, in 1996, and Ph.D. degree in computer architecture from Institute of Computing Technology (ICT), Chinese Academy of Sciences (CAS), Beijing, in 1999. Now he is a professor and Ph.D. supervisor of ICT, CAS, Beijing. His research interests include compiler optimization and binary translation.