M.ENG. THESIS

Competition-Based Adaptive Caching for Out-of-core Graph Processing

외부 그래프 처리를 위한 경쟁 기반의 적응형 캐시 설계

2021 년 2 월

서울대학교 대학원

응용공학과 응용공학전공

명 기 현

Competition-Based Adaptive Caching for Out-of-core Graph Processing
외부 그래프 처리를 위한 경쟁 기반의 적응형 캐시 설계

지도교수 염 헌 영

이 논문을 공학전문석사 학위논문으로 제출함

2020 년 12 월

서울대학교 공학전문대학원

응용공학과 응용공학전공

명 기 현

명기현의 공학전문석사 학위논문을 인준함

2020 년 12 월

위 원 장 (인)

부 위 원 장 염헌영. (인)

위 원 (인)

Abstract

A graph engine should possess adaptability to ensure efficient processing despite a variety of graph data and algorithms. In terms of out-of-core graph engines, which exploit a hierarchical memory structure, an adaptive caching scheme is necessary to sustain effectiveness. A caching policy selectively stores data likely to be used in the upper-layer memory based on its own expectation about the future workload. However, the graph workload contains a complexity of memory access according to graph data, algorithm, and configurations. This makes it difficult for a static caching policy to respond to the changes in workload. In this paper, we propose a graph-adaptive caching scheme which ensures consistent effectiveness under the changing workloads. Our caching scheme employs an adaptive policy that responds to changes in real-time workloads. To detect the changes, we adopt the competition procedures between two contrasting properties—locality and regularity—that appear in graph workloads. In addition, we combine two window adjustment techniques to alleviate the overhead from competition procedures. The proposed caching scheme is applicable to different types of graph engines, achieving better efficiency in memory usage. Our experimental results prove that our scheme improves the performance of graph processing by up to 65% compared to existing schemes.

Keywords: out-of-core graph processing; adaptive policy; memory replacement; page cache; optimization

Student Number: 2019-29953

Contents

Abstract **1**

1 Introduction **6**

2 Background and Motivation **9**

2.1 Memory usage in out-of-core graph engines 9

2.2 The effectiveness of caching in graph engine 10

2.3 Challenges in caching policy selection 12

3 Design and Implementation **14**

3.1 Overview of GAC 15

3.2 Competition mechanism 16

3.2.1 Evaluation rule 17

3.2.2 Active Policy and Fallback Policy 17

3.2.3 Evaluation and state transitions 18

3.3 Competition window 20

3.3.1 Fine-grained grouping 21

3.3.2 Voter-centric competition 21

3.4 Optimization techniques 23

3.4.1 Time decay 23
3.4.2 Soft LIFO 24
3.4.3 Boundary adjustment 24

4 Experimental Evaluation **26**
4.1 Adaptive Policy vs. Static Policy 28
4.2 Overhead 30
4.3 Impact on Overall Performance 32

5 Related work **36**

6 Conclusion **38**

초록 **42**

List of Figures

2.1 The hit ratio while performing the different graph algorithms with varying cache sizes. 11

2.2 Different workloads of the PR and TC algorithm. 12

3.1 The two rules to evaluate the suitability of the two competing policies. 18

3.2 The evaluation mechanism of GAC (below), and state transitions of the page (above) accordingly. 20

3.3 The voter-centric competition and propagation procedure (*A*: Active policy / *F*: Fallback policy). 23

4.1 Twitter dataset (X-axis: cache size / Y-axis: hit ratio) 29

4.2 Friendster dataset (X-axis: cache size / Y-axis: hit ratio) 30

4.3 The real-time score log (Red: CLOCK / Blue: LIFO) for different workloads. 31

4.4 Performance and read volume of GAC compared to FlashGraph (X-axis: cache size / Y-axis: normalized value) 33

List of Tables

4.1 Graph algorithms. 27
4.2 Graph datasets. 28
4.3 Overhead of GAC. 32

Chapter 1

Introduction

The graph engine deals with the high irregularities from real-world graphs and different graph algorithms. Many engines have reacted to the various workloads based on monolithic memory structure, which demands in-memory configuration. However, the increasing graph size makes the design of the in-memory graph system complex and costly. Scientific discovery (e.g. neuroscience) [1] or growing the inner part of graph data (e.g. users and interaction in social graph) [2] requires scalability to handle large graphs having billions of vertices. To accommodate such large graphs with an existing in-memory scheme, the graph engine has to deal with extra issues for distributed processing. Accordingly, an out-of-core graph engine, which allows a single node to spatial scalability, has been actively considered as an alternative or complementary choice.

The out-of-core graph engine utilizes an external storage device, which is relatively slower but provides higher capacity than RAM. The entire graph dataset is stored as a file in external storage. As the graph engine traverses the dataset in algorithmic order, the required data are partially loaded into memory space. Existing studies have shown

a semi-external engine and external engine. The former stores only the edge lists in external storage, whereas the latter stores all types of data in external memory including vertex state, index table, and edge lists. We collectively refer to both methods as an out-of-core graph engine in this study.

In the out-of-core graph engine, the page cache stores some parts of the entire graph in memory, so that those parts are available without accessing the external storage. To achieve high efficiency of the cache, a page replacement policy plays a key role, keeping the selective entries that are likely to be used in the near future. The policy is mainly based on the heuristics of the future memory workloads. When the cache needs a room for the newly loaded page, it must discard a page not likely to be used in the near future (i.e., to be used in the distant future).

For graph workloads, we focus on two replacement policies: CLOCK and Last-in, First-out (LIFO). In general-purpose systems, including linux kernel [3], CLOCK is commonly employed based on the assumption that some pages have temporal locality. CLOCK checks the reference flag of each page by iterating clockwise. Then, the first page found that has not been used between two accesses of clockhead is selected as a victim. Conversely, LIFO discards the recently used page. For programs that have looping reference patterns, LIFO is known to be stable. In this paper, we define a property causing this access pattern as regularity.

Graph processing is a special application that contains two opposite properties of memory access, regularity and locality. Many graph algorithms have regularity due to their vertex-centric programming model [4], where all activated vertices are traversed multiple times in a similar order. At the same time, in terms of locality, many real-world graphs follow a power-law distribution. So, the minority of vertices have extremely high degree. The graph edge traversal algorithms tend to intensively access small memory areas that contain the vertex data with high vertex degree. Existing studies [5, 6] support several caching policies, so that the users choose different policies

depending on the graph workload. This method is sometimes impractical if the memory access pattern is unpredictable. Another study pre-selects important data through pre-processing [7], but has difficulty coping with dynamic changes in graph algorithms and datasets.

In light of these limitations, out-of-core graph engines need an adaptive solution that automatically selects a suitable policy for real-time workloads.

We propose graph-adaptive caching (GAC) to help out-of-core graph engines provide high effectiveness against a changing workload. To accomplish this goal, we simulate two competing policies—LIFO and CLOCK, and then evaluate the simulation results on the basis of actual accessing events. This methodology focuses on offering better effectiveness of the page replacement decision compared to using a static policy.

For further optimization, we add a fine-grained grouping and a voter-centric competition scheme. They realize the lightweight caching mechanism. The grouping technique reduces the lock contention, and promotes the competition procedures by finely adjusting the access window of the threads. With a voter-centric competition scheme, voter groups crown one policy as a global policy and follower groups abide by the global policy. Additionally, there are three optimization techniques—time decay, soft LIFO, and boundary adjustment—to be introduced.

Among the out-of-core graph engines, we developed GAC on top of FlashGraph [6, 8], which is one of the state-of-the-art graph engines. Through our experimental results, we prove that GAC significantly improves hit ratio, reduces the number of reads, and boosts overall performance accordingly. To our knowledge, this is the first approach to dynamically reflect the characteristics of graph workloads into the caching scheme.

Chapter 2

Background and Motivation

2.1 Memory usage in out-of-core graph engines

Graph engines for large-scale graph dataset are classified into distributed graph engines (in-memory) and out-of-core graph engines. Most graph engines that adopt a distributed method [4, 9, 10] need large memory capacity across multiple servers and network infrastructure. The related studies have focused on synchronization and load balancing techniques to optimize the distributed structure. Alternatively, large graphs can be handled in a single node by using an out-of-core processing method.

There are three main types of data utilized in out-of-core processing: vertex state, edgelist, and index table. The vertex state stores the intermediate state of the vertices while performing the algorithm. Depending on the algorithm's purpose, the state keeps different information. For example, in a weakly connected components (WCC) algorithm, the vertex state holds the smallest ID among the IDs delivered from neighbors and itself. These values can be stored in either external storage [11, 12] or internal memory [6, 13] according to the type of engines (external or semi-external). The edge

list keeps the ID list of the neighboring vertices. The out-of-core engines normally manage the edge list with external storage. The index table helps a graph engine find out the external location (file ID and file offset) by referring the mapping information from vertex ID to page location.

We emphasize workloads for the edge list since it accounts for most of the graph data. The edge lists are retrieved as vertex ID granularity (4 B or 8 B) after being loaded into memory, but managed as page granularity (4 KB or 8 KB) in external storage. This makes the page access pattern more irregular. To streamline the accessing order and units, existing graph engines have transformed the layout of edge lists to customize it for their operational method. However, in terms of memory access patterns, such customization increases the irregularity. Even if the same dataset and algorithm are utilized, the memory access order can be completely different depending on the graph engine's implementation. We introduce their major techniques in Section 5.

2.2 The effectiveness of caching in graph engine

Caching greatly affects the data access speed in the hierarchical memory system. With a highly effective cache, data are accessed with the high speed of the upper layer while making use of the large capacity of the lower layer. Likewise, since the out-of-core graph engines exploit both memory layers—internal memory and external storage—the usage of caching is critical. The effectiveness of a cache depends on the suitability of the caching policy to the workload. Although it is common that the hit ratio increases along with the increasing cache size, it may not be if the caching policy is unsuitable for the workloads.

Figure 2.1 demonstrates the effectiveness of the page cache while performing various graph algorithms with a Twitter dataset [14]. Under the CLOCK policy, the pages that have not been recently used are discarded. As shown in the figure, PageRank (PR)

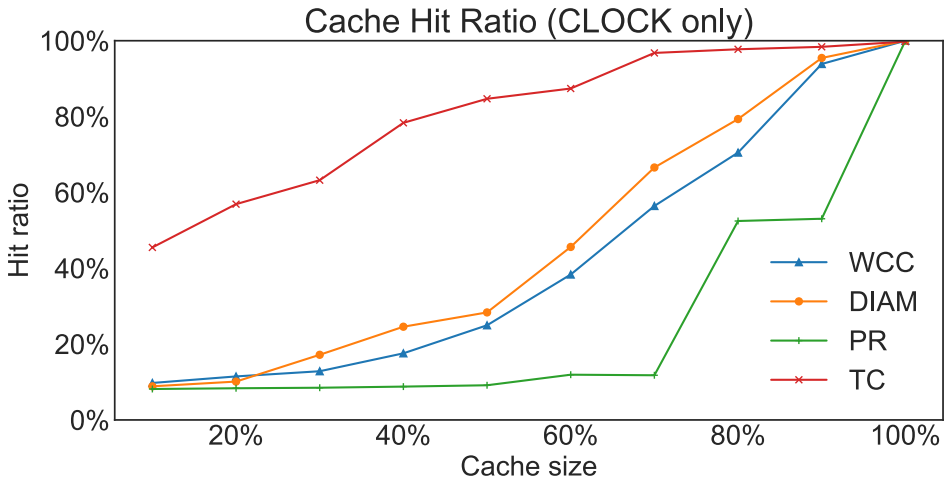
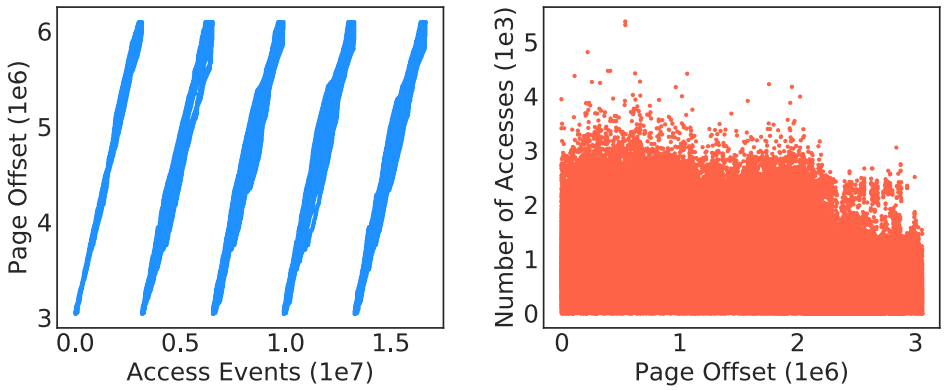


Figure 2.1: The hit ratio while performing the different graph algorithms with varying cache sizes.

and triangle count (TC) provide significant disparity in cache effectiveness. The hit ratio for PR hardly increases until 70% capacity of the dataset is used as cache size. On the other hand, in the case of TC, a high hit ratio of 40% or more is continuously achieved. The WCC and diameter estimation (DIAM) algorithms show moderate effectiveness between PR and TC.

The disparity results mainly from the different access order of the algorithms. Figure 2.2 shows the different memory workloads of PR and TC with a Twitter dataset [14]. Since TC and PR are implemented to refer only to in-edges and out-edges, respectively, the page offset does not overlap with each other in Fig. 2.2a and Fig. 2.2b. The PR algorithm regularly accesses the edge lists of the activated vertices in their ID order. This results in a looping access pattern to large memory area as shown in Fig. 2.2a. Meanwhile, the TC algorithm counts the number of triangles, which is a set of three neighboring nodes. Therefore, the edge list of neighbors are indirectly referenced to form the potential triangle. Consequently, as shown in Fig. 2.2b, the small number of pages are referenced thousands of times, whereas the reference count for all pages



(a) Page accessing pattern of PR

(b) Page reference count of TC

Figure 2.2: Different workloads of the PR and TC algorithm.

is an of average 312.

If CLOCK is used for PR, most of the cache entries are thrashed within an iteration, so the entries are never reused in the subsequent iteration. This produced the ineffectiveness of CLOCK in Figure 2.1. On the other hand, CLOCK achieved high effectiveness for TC by keeping the pages with temporal locality in the cache entries. Our observation here draws on motivation that using a static caching policy for graph engines cannot preserve high effectiveness against different workloads. Considering the large disparity between PR and TC, graph caching should reflect both the repetitive nature of the graph algorithms and the power-law distribution of the graph dataset. Our scheme projects both regularity and locality onto the LIFO policy and CLOCK policy, respectively, and makes the aggregated decision.

2.3 Challenges in caching policy selection

There are existing studies that react to various access patterns of the graph application. One way is to provide a variety of cache algorithms, allowing the users to choose which one is most suitable for their tasks. GC [5] and GraphCache [15] identify the

differences in trade-off of the cache replacement policies under the graph workload and dataset. Hence, GC offers four cache policies, for users to choose from that take the lead depending on the conditions.

The other way is to pre-analyze the expected workload based on the algorithm and dataset offline. BASC [7] reallocates vertex IDs following the access order of the breadth-first search (BFS) algorithm. After that, it pre-selects and holds the static cache entries in order of probability that each page will be traversed while performing the BFS-like algorithms. The probability is calculated by reflecting the distribution of the dataset, such as the number of reverse-edges of each vertex. This is to counter the regular access pattern of the graph algorithms with a static cache, and to prepare for locality by analyzing the graph dataset.

While both of the above methods rely on the offline analysis, graph workload indeed has explicit non-determinism [16]. As long as the final result meets the acceptance criteria, one algorithm can be implemented in several ways. The memory access pattern also depends on a variety of system configurations, such as multi-threading, memory size, and dataset. In multi-threading, modern graph engines, which follow the scatter-apply-gather paradigm [4], generate non-deterministic workloads with concurrent accesses to independent partitions. The different memory size sometimes requires different cache policy according to the cache size compared to the working memory size. Additionally, the graph dataset keeps changing in the real world by adding or removing the vertices and edges. Thus, it reduces the usefulness of pre-selected entries [7] and pre-selected policy [15]. On the contrary, our caching scheme tackles changing workload based on its real-time evaluation method without any offline phase.

Chapter 3

Design and Implementation

GAC is a pluggable cache, allowing any type of graph engine to be incorporated. It aims to realize both the adaptability to graph workloads and lightweight caching. The adaptive caching policy reduces the number of accesses to external storage, leading to performance improvement. To this end, we design GAC with the following principles:

Adaptability: For access patterns that some pages are repetitively referenced over a short period of time, GAC adopts CLOCK; for patterns indicative of how most pages are regularly accessed, it adopts LIFO. Thus, GAC provides the best of both worlds by determining the better policy in real-time.

Performance: If time complexity of the cache algorithm is too high, the acquired adaptability becomes useless. Thus, GAC must assure a comparable or better performance than the use of a static policy. After all, we compare the overall performance before and after applying GAC.

Algorithm 1 Overview of the GAC procedure

```
 1: procedure SEARCH_CACHE(offset, callback)                    ▷ Cache class
 2:     group = find_group_by_offset(offset)
 3:     ret = group→search_group(offset)
 4:     if ret == NULL then
 5:         page = group→get_empty_page(offset)
 6:         async_load_request(page, offset, callback)
 7:     end if
 8:     return page
 9: end procedure

10: procedure GET_EMPTY_PAGE(offset)                            ▷ Page group class
11:     if this→is_voter() then                                 ▷ For voter groups
12:         evaluate_ghost(offset)
13:         clock_vict = voter_clock_simulate()
14:         lifo_vict = voter_lifok_simulate()
15:         actual_victim = evict_victim(clock_vict, lifo_vict)
16:         global_update(this→local_score)
17:         time_decay()
18:         return actual_victim
19:     else                                                    ▷ For follower groups
20:         if global_load() == CLOCK then
21:             victim = clock_evict(this)
22:         else
23:             victim = lifo_evict(this)
24:         end if
25:         return victim;
26:     end if
27: end procedure
```

3.1 Overview of GAC

GAC is a user-level page cache that handles all access to external data with bypassing the kernel page cache. If a graph engine requests certain data with an external memory address (file offset), GAC returns an internal memory address (memory reference) on which the needed data is located. GAC can be incorporated into different engines by connecting the engine's I/O interface with the GAC.

Algorithm 1 shows the overall procedures after a specific page is requested to GAC. First, a page group, to which the requested page belongs, is obtained (line 2).

By traversing the entries inside the group (line 3), it processes whether the page currently exists in memory or not. If an entry matching the requested offset is found, (i.e., cache hit), GAC returns the reference (line 8). Otherwise (i.e., cache miss), it has to reclaim one page to make room for the newly loaded page (line 5). To mask the latency of storage device, we borrow the optimization technique from SAFS [8]—an asynchronous user-task I/O interface—that transfers the graph algorithm as a callback function (line 6). This method also prevents the additional memory copy from the I/O buffer to the computation buffer.

The page eviction functionality (line 10) is crucial to the memory efficiency. In Section 2, we discussed how the existing caching solutions are ineffective and why the adaptive policy is necessary in graph processing. To decipher the changing workload in real-time, we apply a competition-based replacement policy. When a cache miss occurs, the two competing policies—CLOCK and LIFO—simulate their victim selection process (lines 13-14), traversing the cache entries in its own way. Then, the actual victim is selected according to active policy—the policy regarded as more suitable for the current workload. In the remainder of this section, we describe the internal implementation of each procedures in Algorithm 1.

3.2 Competition mechanism

In this subsection, we describe how GAC manages the competition mechanism between the two policies. First, we justify two rules that are the foundation for evaluating the suitability of each policy. The two policies are classified into an active policy and a fallback policy according to the cumulative competition results. Thus, we introduce how each policy works for selecting the victim. Finally, we illustrate the evaluation procedure for each page and the state transition, accordingly.

3.2.1 Evaluation rule

Figure 3.1 depicts the underlying rules for evaluation. In the figure, we regard the competing policies as anonymous *A* and *B*, and ***0x1*** indicates the unique page number. The timestamp *T* is used for as a decay function that is introduced in Section 3.4.1. To describe the evaluation process, we first define the following rules:

Rule 1 If ***0x1*** is selected earlier by *A*, then selected later by *B* without being accessed in between, *A* is considered the better policy.

Rule 2 If ***0x1*** is selected earlier by *A*, but ***0x1*** is accessed before *B* selects, then *B* is considered the better policy.

The above rules are partially based on Belady's algorithm [17] (also known as OPT). This algorithm evicts a page to be used in the most distant future by reflecting the future access order. In practice, it is considered infeasible because the future access pattern cannot be known beforehand. Therefore, this method is mainly used offline to evaluate other replacement algorithms. To extend this idea to the online method, we evaluate the policies by looking back at their past prediction (selection history) when the prediction is disambiguated. With the comparison of the two policies based on the rules, GAC can determine the better policy that makes the re-fault distance longer.

3.2.2 Active Policy and Fallback Policy

According to the above rules, the competition results are stacked as a score. The policy score is updated in the positive direction when LIFO wins and in the negative direction when CLOCK wins. The policy that gains more score is set as the active policy; the other policy becomes the fallback policy. In other words, LIFO is set as the active policy when the score is positive; CLOCK is set as the active policy when the score is negative. We set LIFO as the starting policy—the active policy when the score is neutral (zero), by default. Depending on the state of each policy, —active or fallback—the

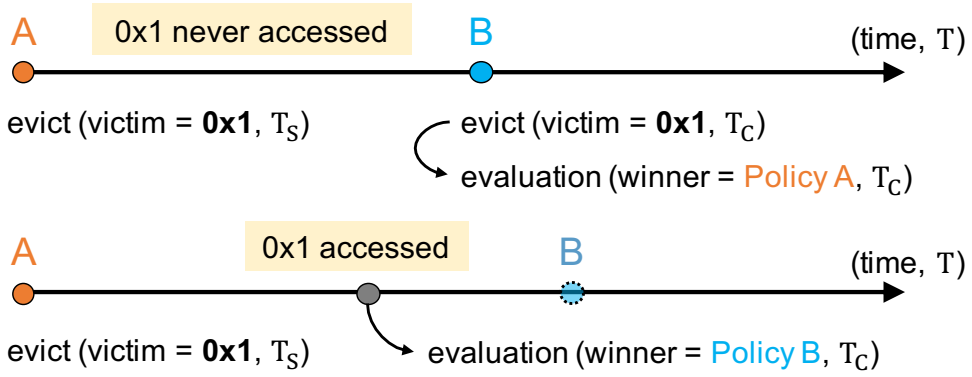


Figure 3.1: The two rules to evaluate the suitability of the two competing policies.

victim selection makes different effects in the cached pages. The page entry selected by the active policy is physically evicted and overwritten with the newly loaded page. On the other hand, if the page entry is selected by the fallback policy, it is only tagged (i.e., virtually evicted).

3.2.3 Evaluation and state transitions

Figure 3.2 depicts the state transition of each page according to competition events. Here, we indicate the active policy and fallback policy as *A* and *F*, respectively. CLOCK and LIFO become either *A* or *F*, depending on the score. At the starting point of the graph algorithm, all pages are placed in external memory, which is "not cached" in the figure. When a cache miss occurs, the page is loaded into memory space (cache entry) and the state becomes "cached." The following describes how the competition process works after the page is cached and cache entries are full.

Cached ↔ Evicted (Ghost): If the cached entry is physically evicted by the active policy, the page becomes a ghost page. Then, the properties in memory are completely overwritten with those of the new page. In this "evicted" state, the data copy exists only in external storage. To proceed with the evaluation rules of this state, the GAC

records the eviction history as a list of ghost pages. The ghost page only keeps the metadata of the evicted page, such as file offset, eviction policy, and timestamp. This list is traversed right after the cache miss (Algorithm 1, line 12). If the page offset is found in the ghost list, the eviction policy is penalized according to Rule 2. This corresponds to *F-2* in the table. Hence, the fallback policy wins by not selecting the page yet. Then, the ghost page becomes "cached" again as the corresponding page is loaded from external storage. In our implementation, we utilize a limited number (16 for each group) of ghost pages.

Cached $\leftrightarrow$ Tagged: When the fallback policy selects a victim, we mark the page state as "tagged". The victim remains in memory even after being selected. In this case, the page entry keeps the selection history, such as the eviction flag, eviction policy, and timestamp. GAC identifies whether the entry is in the "tagged" state by checking the eviction flag. While the policies simulate their victim selection (line 13 and 14), the reclamation head checks the eviction flag in a *test_and_clear()* manner. If it turns out that an entry has been accessed since it was tagged, GAC judges that the fallback policy has made the wrong decision (Rule 2). The active policy wins at this competition (*A-2*).

Evicted / Tagged $\rightarrow$ Not cached: If the competition process ends according to Rule 1, the page enters a "not cached" state. Since the page was not accessed between the choices of the two policies, the preceding policy becomes the winner in *A-1* and *F-1*. This is because the preceding correct choice will make the re-fault distance longer. This evaluation process is handled inside the simulation functions (line 13 and 14), and mainly inside the eviction function (line 15) where two simulation results are collected. When the ghost list is full in the "evicted" state, the oldest ghost page is discarded, and the page state returns to "not cached."

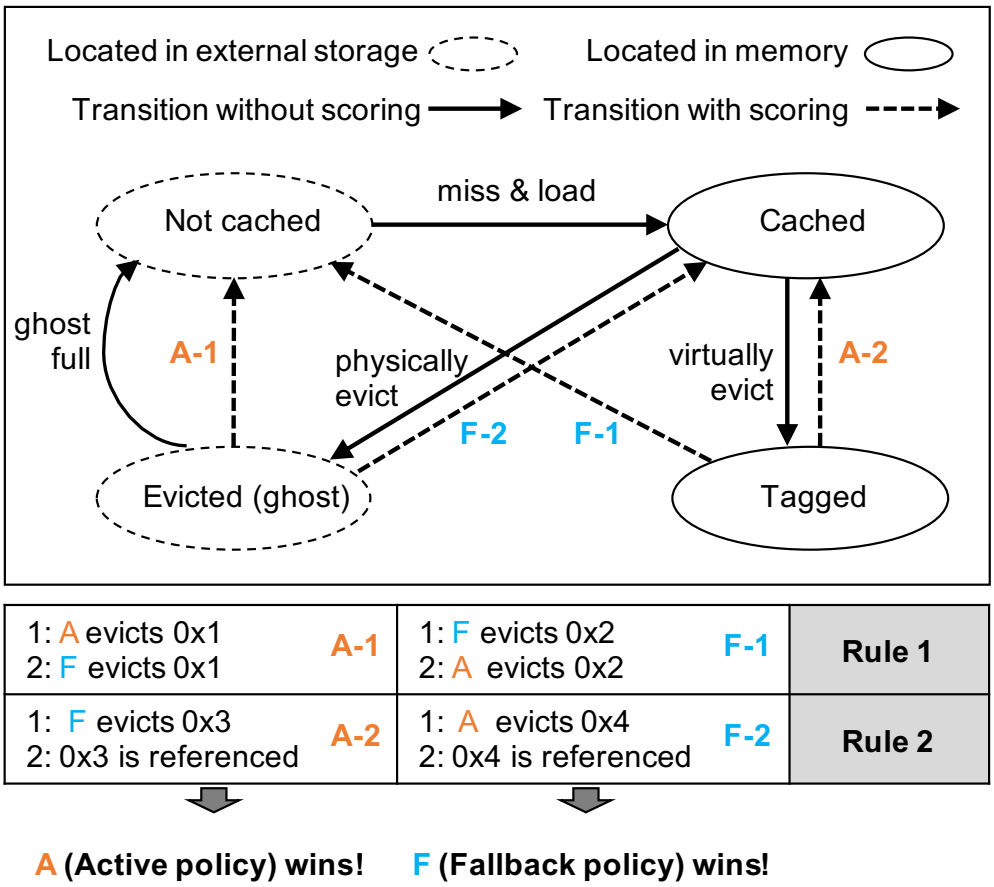


Figure 3.2: The evaluation mechanism of GAC (below), and state transitions of the page (above) accordingly.

3.3 Competition window

In the previous section, we described how GAC adapts to workloads in real-time. Here, we focus on the implementation techniques to minimize the competition overhead. This is composed of two techniques, fine-grained grouping and voter-centric evaluation.

3.3.1 Fine-grained grouping

The modern graph engine exploits the high parallelism. If the replacement process (Algorithm 1, line 5) takes too long, the locking contention of multiple threads for cache access (line 1) becomes severe. In addition, if the traversal window is too wide, Rule 1 does not work properly because the choices of CLOCK and LIFO are rarely overlapped. In this case, the cache effectiveness greatly relies on the starting policy.

Fine-grained grouping shrinks the access window of the competition mechanism into a group of sixteen cache entries. All pages of the graph file are finely associated with a certain group after processing 2-level hashing (line 2). Since the mapping to the page group is handled in a deterministic manner by calculating a hash function, each page is searched on the associated group to which the page belongs. If the page does not exist in that group, a victim selection process is initiated within the group window (line 5).

This method localizes the competition procedures, so that the results infer the access patterns within the page groups. Alternatively, page groups can be organized by binding the correlated pages with subgraph extraction techniques [18]. However, for this method, to separate the independent vertices that belong to same page, preprocessing is inevitable. This part is beyond the scope of this paper. We leave it to future work. Each group can be associated with the variable number of entries. However, this number invokes the trade-off issue often cited for the CPU cache design—conflict miss (if too small) and cache traverse time (if too large) [19]—so it is recommended to maintain an appropriate level.

3.3.2 Voter-centric competition

This grouping technique relieves the lock contention and facilitates evaluation with the fine-grained window. In return, this requires overhead to simulate both policies—

CLOCK and LIFO—in all groups. We improve the competition efficiency by managing global score on the basis of the voter-centric competition. In the grouping mechanism introduced earlier, each group locally maintains its own score and policy. When adding the voter-centric evaluation, only voter groups manage the shared global score. Each group is randomly classified as either a voter or follower at the initialization stage of GAC. The selected voter groups still perform the full competition process (lines 11-18), but the others, designated as followers, are not involved in the competition process (lines 19-25).

Figure 3.3 shows an overview of voter-centric competition. When a thread enters a voter group by address hashing, it refers to the competition history within the group. Then, if the score changes during the simulations, the delta is updated into the global score (line 16). Since multiple threads are concurrently working on different voter groups, the global score may create a race condition. Therefore, the working threads update the score using an atomic operation such as *compare_and_swap*. If the thread enters a follower group, it loads the global score (line 20) and executes the corresponding policy. As a result, the follower groups continue to pull information from the voter groups.

This mechanism makes sense because the policy selection corresponds to an NP-hard problem. If GAC can identify and collect a suitable policy for the random page groups, GAC can reason its suitability for the overall memory region. Even though the competition overhead is concentrated into voter groups, all threads evenly share the cost because they access either voter group or follower group according to page address. We used 1000 page groups as the voters, that is, these are entries of about 64 MB (4 KB page$*16*1000$) in size. Note that it is a binary-decision problem, in which GAC expects the better one of the two, not the continuous value. Since the higher number of voters does not improve the hit ratio, we keep it fixed in our experiments.

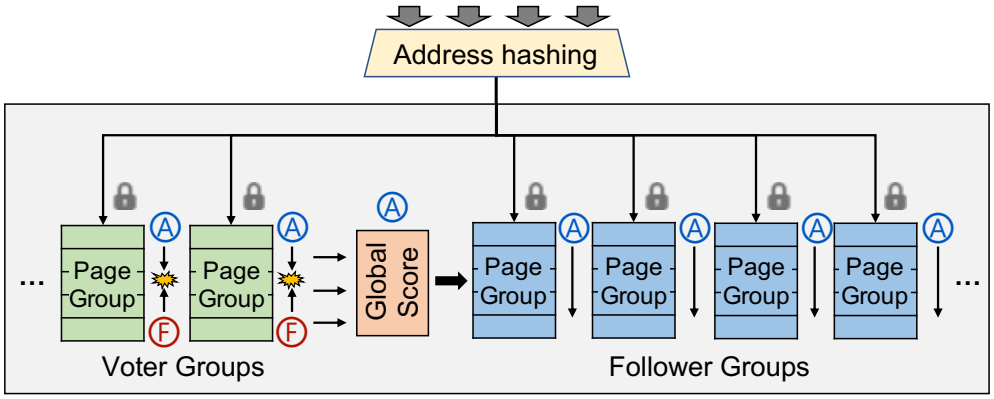


Figure 3.3: The voter-centric competition and propagation procedure (*A*: Active policy / *F*: Fallback policy).

3.4 Optimization techniques

We introduce three optimization techniques—time decay, soft LIFO, boundary adjustment—to improve the high effectiveness of our caching policy. Time decay allows our scheme to quickly adapt to changes in the algorithm. Soft LIFO and boundary adjustment are employed to resolve the problem of group sharing and page sharing, respectively, in a parallel processing environment.

3.4.1 Time decay

Even while performing one algorithm, the workload changes over time. For example, the memory region to be accessed is drastically reduced if many vertices of the dataset are converged after several iterations. In such case, the recent competition events contain more information about the future workload. Thus, we weight the recent information by applying exponential decay.

As shown in Figure 3.1, we check the timestamp each time the competition starts and ends. When a winner policy is disclosed, the score is updated by adding or subtracting one point by default. When applying the decay function, the score is updated

while decaying to $1*d^{(T_C - T_S)}$ in the evaluation function of Figure 3.1. In this equation, d is the decay factor, for which we fixedly used 0.7, and it adjusts how sensitively our scheme responds to changes in workload. S and C correspond to the starting time and completion time, respectively, of each competition. The global score is also decayed as T increases (line 17).

3.4.2 Soft LIFO

In Section 2.1, we pointed out the difference in accessing granularity between the edge list and the external storage block. Occasionally, a page is accessed multiple times even with looping access, as the page holds the edge lists of multiple vertices together. When two threads alternately access a group (i.e., with group sharing), two threads may interfere with each other. For example, the subsequent thread removes the most recently used page, which is possibly in use by the preceding thread. This interference not only leads to nested misses between the threads, but LIFO also continues to lose scores despite the looping workload. Thus, we utilize a soft LIFO policy, which discards the second-last page instead of the last. This method resolves the interference caused by multiple threads sharing one group. On the surface, soft LIFO evicts the two most recent entries, alternately.

3.4.3 Boundary adjustment

Boundary adjustment copes with the page sharing problem of two threads. In the vertex-centric programming model, partition size is commonly based on the number of vertices, and the edgelists are contiguously stored in vertex ID order. Therefore, two partitions share one boundary page except for the case when the partition boundary is aligned by chance. Even with completely looping access, the boundary pages are accessed twice for different partitions. Hence, GAC preserves the boundary page as one

of the static LIFO entries until both threads reference it. We refer to the index table in the graph engine to identify the boundary page from vertex ID to page offset. To realize this, LIFO reclamation head checks the boundary_bit when it chooses a victim (Algorithm 1, lines 14 and 22).

Chapter 4

Experimental Evaluation

In this section, we experimentally prove the effectiveness of GAC. Firstly, we verify whether the GAC follows well one of the two static policies, CLOCK and LIFO, against the changing graph workloads. The workload is diversified by varying graph algorithm, graph dataset, and configurations (cache size). Next, we demonstrate the potential improvement that can be achieved by switching the cache policy in real time. We also offer the measured overhead incurred by managing the adaptive replacement policy. Lastly, we present the execution time to finish the graph algorithms compared to before applying the GAC. We use 16 threads for all experiments, so that the parallelism causes additional irregularities as explained in Section 2.3.

We mainly implement and test GAC on FlashGraph. We choose FlashGraph because this is one of the representative out-of-core graph engines that provides high compatibility with the upper layer such as the R base package. Also, this engine is open-source and has been studied by many subsequent research groups. GAC is applied as an edge list cache of this engine. Since FlashGraph has already incorporated its own caching scheme with the CLOCK policy [8], we prove how beneficial applying

Table 4.1: Graph algorithms.

Algorithms	Description
Weakly Connected Component (WCC)	Finding sets of connected nodes in the undirected graph
Diameter Estimation (DIAM)	Finding the largest distance between any pair of nodes
PageRank (PR)	Estimating the importance of each node in the directed graph
Triangle Count (TC)	Determining the cohesiveness of the three nodes

our scheme is by replacing it with ours.

Four graph algorithms—WCC, DIAM, PR and TC— are targeted in our experimental results, as presented in Table-4.1. In addition, we use two real-world graph datasets presented in Table-4.2. The raw graphs are initially compressed according to the encoding format of the graph engine. More algorithms, such as strongly-connected components and BFS, with more datasets were also experimented, but we do not present the results here as the results do not deviate much from the results presented here.

All experiments were performed on a single machine with Intel Xeon E7-8870 v3 having 256 GB memory. Of the entire memory, the cache size was adjusted in proportion to the size of the dataset. In terms of performance, storage bandwidth also affects the overall time to run the algorithms. Thus, we employed two different types of storage device—NVMe SSD and SATA SSD—for performance comparison. Only the read bandwidth of SSDs affected the performance for loading the missed pages into the cache memory. NVMe SSD offers 3 GB/s read and SATA SSD offers 550 MB/s read. The OS page cache does not have an effect on the performance since all I/O requests are coordinated through Direct I/O.

Table 4.2: Graph datasets.

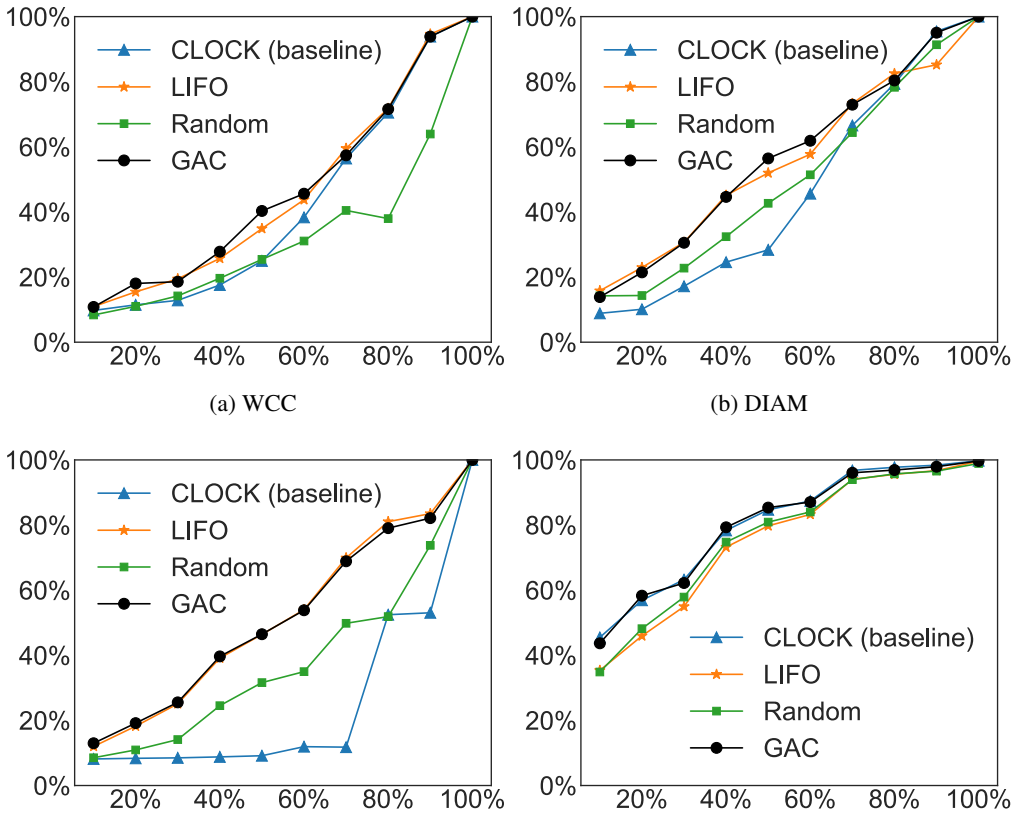
Graph Dataset	# vertices	# edges	graph size
Twitter (TW) [14]	41M	1.46B	25GB
Friendster (FR) [20]	65M	1.80B	31GB

4.1 Adaptive Policy vs. Static Policy

Figure 4.1 and 4.2 show the changes in hit ratio according to graph datasets and algorithms, and cache size. The cold misses are excluded from calculating the hit ratio. The cache size is also considered for the hit ratio because it causes the group allocation to be different depending on the hash result. We compared GAC against three static algorithms—CLOCK, LIFO, and Random. The Random policy is mobilized to show the ineffectiveness of the unsuitable policy.

As we have explained so far, type of algorithm is a crucial part for effectiveness of the caching policy. In particular, LIFO for PR and CLOCK for TC show a better hit ratio according to their opposite approach patterns as demonstrated in Section 2.2. For both algorithms, GAC exhibits a comparable hit ratio to the better policy by adapting to the graph workloads. It offers a significant advantage over the use of an unsuitable policy by up to 57% in hit ratio (the 70% size in Figure 4.1c). Even with an identical algorithm, the effectiveness of a certain policy varies depending on the dataset. For example, if most vertices remain activated after several iterations, this exposes the ineffectiveness of using CLOCK as shown in Figure 4.1c. In the example of Figure 4.2c, the CLOCK hit ratio is greater than 4.1 as more vertices converge over several iterations.

As shown in Figure 4.2d, the use of a static LIFO policy may produce an unstable hit ratio. This is because LIFO dismisses the different locality of the pages. If LIFO keeps holding pages that are not reused, the hit ratio may not be improved even with the larger cache size. In this case, the effectiveness is influenced by the initial layout



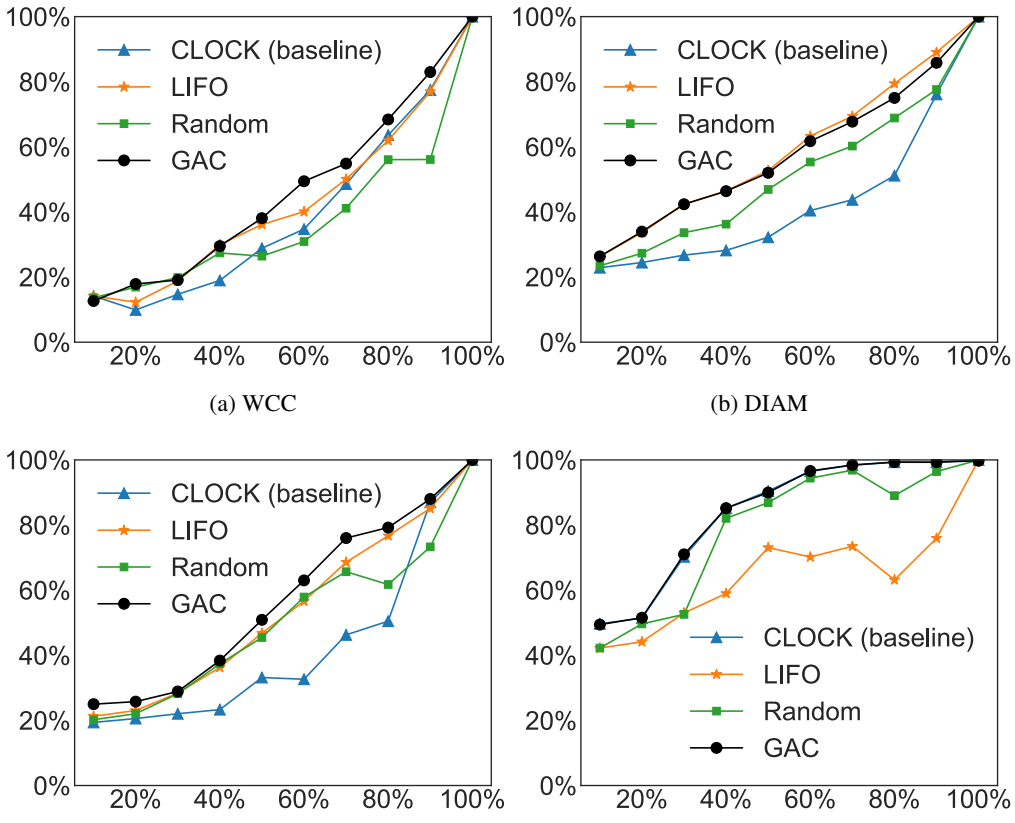
(a) WCC (b) DIAM

(c) PR (d) TC

Figure 4.1: Twitter dataset (X-axis: cache size / Y-axis: hit ratio)

of the graph dataset.

For WCC and DIAM algorithms (in Figure 4.1 and 4.2), the gap between LIFO and CLOCK is relatively small. This is because the two properties of the graph workload—regularity and locality—are reflected as a mixture. Figure 4.3 represents the real-time score log in the different configurations. In both cases, the GAC keeps track of the changes in graph workload based on the competition results. Then, the active policy is changed according to the score. The decay function prevents the score from exploding and allows the GAC to quickly adapt to the changes. As a result, at those points, GAC outperformed LIFO, which is the better policy, by 5.5% and 9.35%, respectively. This



(a) WCC (b) DIAM

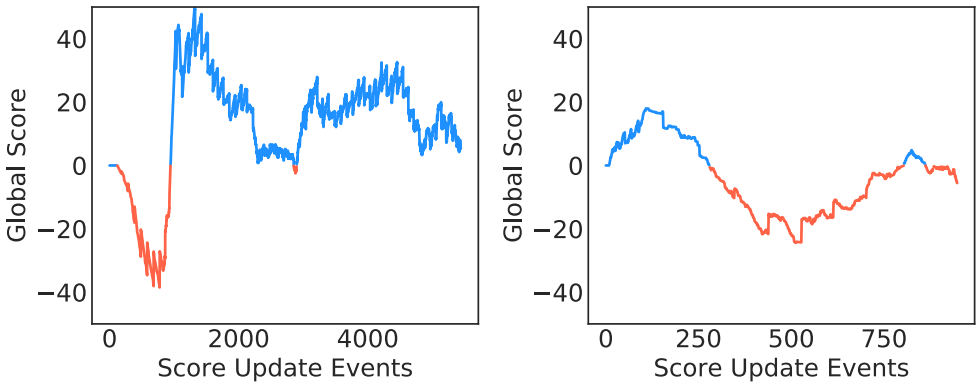
(c) PR (d) TC

Figure 4.2: Friendster dataset (X-axis: cache size / Y-axis: hit ratio)

demonstrates that potential improvement can be obtained by switching the two policies in real time like a hybrid policy.

4.2 Overhead

Table-4.3 shows the CPU time to execute the eviction process for each thread. Each function is listed on the basis of the procedures (lines 13-25) described in Algorithm 1. In Table-4.3-a, full competition corresponds to the case that all fine-grained groups manage their own state, such as score and competition logs. Table-4.3-b is the case in which the voter-centric competition is reflected. The overhead ratio for both cases is



(a) TW + DIAM + 50%

(b) FR + WCC + 50%

Figure 4.3: The real-time score log (Red: CLOCK / Blue: LIFO) for different workloads.

calculated compared to the length of CPU time required to run one static policy. In our scheme, the follower group executes only the global policy, so we set the baseline as the time for *basic_policy* of "Follower" in Table-4.3-b.

In case of the full competition, all threads go through the competition process (Algorithm 1, lines 13-18) in any group to identify workload for the group. It leads to overhead of up to 437% in the TC algorithm. In our implementation, simulating LIFO took more time to traverse the victim because it additionally utilizes a LIFO ordered-list, which is less friendly to the CPU cache block. Specifically, GAC spent the most time in *evict*, where the simulation results are aggregated.

When voter-centric competition is applied, the competition process is performed only if a miss occurs on the pages belonging to the voter groups. It significantly reduces competition overhead. Although an update function is supplemented to guarantee the atomic update of threads to the global state, overhead is negligible in our measurement. Follower groups only need to execute one dynamic policy according to the consensus result of the voter groups. As a result, GAC keeps an overhead of less than 6%.

In terms of the space overhead, GAC requires some metadata for normal pages and

Table 4.3: Overhead of GAC.

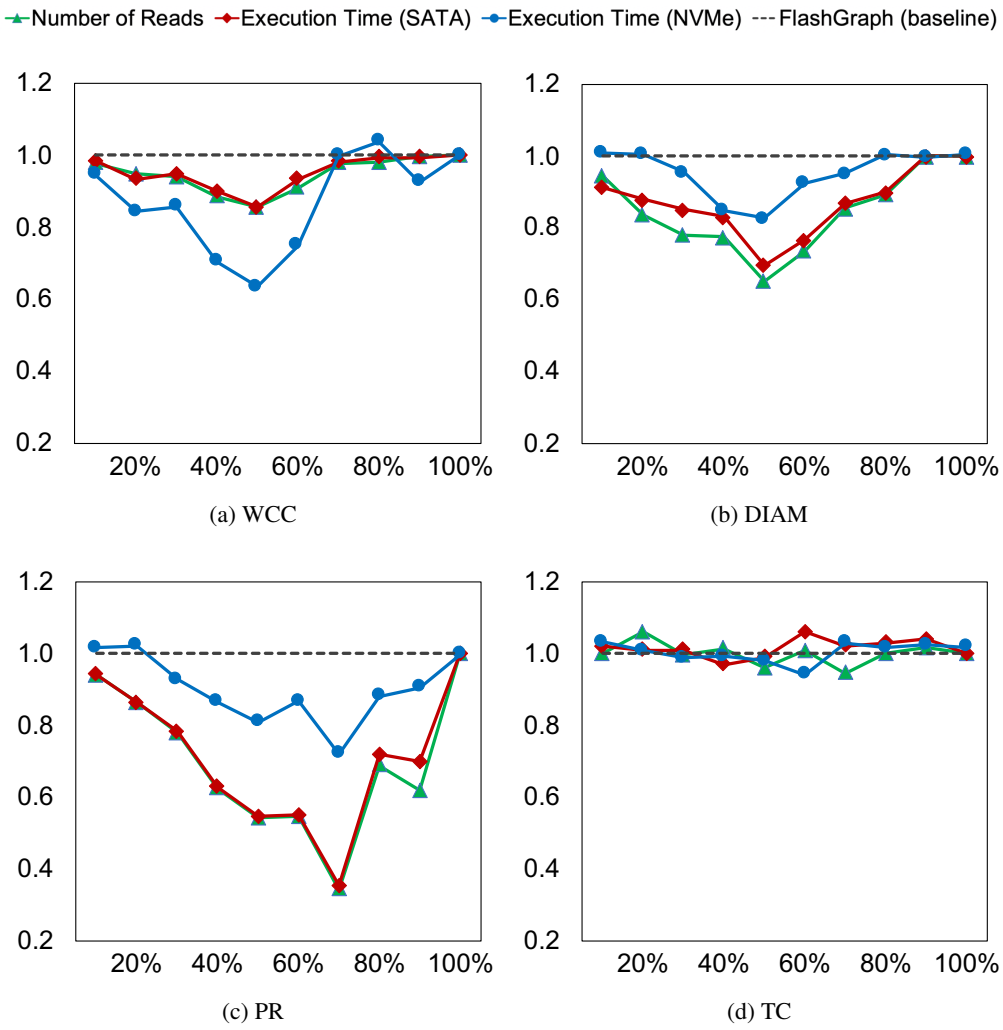
(a) Full competition - CPU time (ms)					
Groups	**Functions**	***WCC***	***DIAM***	***PR***	***TC***
All	*clock_sim*	704	5,195	2,328	81.5K
	lifo_sim	1,853	6,780	4,152	107.6K
	evict	2,450	11,936	5,045	291.3K
	-	-	-	-	-
	decay	260	924	299	9.3K
-		-	-	-	-
Total (ms)		5.3K	24.8K	11.8K	489.7K
Overhead (%)		**165%**	**242%**	**158%**	**437%**

(b) Voter-centric competition - CPU time (ms)					
Groups	**Functions**	***WCC***	***DIAM***	***PR***	***TC***
Voter	*clock_sim*	6	30	41	878
	lifo_sim	8	26	32	941
	evict	13	63	55	3,019
	update	2	10	5	226
	decay	1	4	2	88
Follower	*basic_pol.*	1,988	7,272	4,579	91.2K
Total (ms)		2.4K	5.0K	6.7K	96.3K
Overhead (%)		**1.51%**	**1.72%**	**2.95%**	**5.65%**

ghost pages to define the competition state. For normal pages, GAC consumes 17 B for an entry frame (8 B), the eviction timestamp (8 B), and several flags (bits) such as v-evict, eviction policy, boundary, etc. For ghost pages, GAC uses 16 page frames for each group to keep the metadata of the physically evicted pages. Ghost pages also takes up 17 B to store the page offset (8-B), the eviction timestamp (8 B), and the eviction policy (1 bit). Overall, each voter group utilizes the metadata of 544 B (i.e., 272 B for 16 normal pages and 272 B for 16 ghost pages) for maintaining 64 KB data (16 normal pages). Hence, the space overhead corresponds to 0.83%.

4.3 Impact on Overall Performance

Figure 4.4 demonstrates how GAC affects the number of reads and the execution time. For all experiments, we fixedly used the Twitter dataset. The *X-axis* represents cache



(a) WCC

(b) DIAM

(c) PR

(d) TC

Figure 4.4: Performance and read volume of GAC compared to FlashGraph (X-axis: cache size / Y-axis: normalized value)

size, and and the *Y-axis* corresponds to the observed value for the number of reads and execution time. In the figure, all measurements are normalized to those of FlashGraph. Note that the FlashGraph statically utilizes a CLOCK-family replacement policy (generalized CLOCK) by default. The lower the height of all the bars in the chart, the greater the improvement compared to FlashGraph.

Number of reads: The number of reads correspond to the hit ratio. Therefore, the higher hit ratio of GAC leads to a reduced number of reads in Figure 4.4. For example, at 50% cache size of WCC and DIAM, and 70% cache size of PR, the gap between GAC and CLOCK turns out to be the largest in both Figure 4.1 and Figure 4.4. Although there is a difference of more than six times in the 70% size in Figure 4.1c, the difference in the number of reads in Figure 4.4c is less since the reads during cache warm-up are equally processed across all policies. For the TC algorithm in Figure 4.4d, GAC and CLOCK show comparable results (within 10% difference of each other).

Execution time: Figure 4.4 also shows the performance improvement, which is reflected in the graph applications, by applying our scheme. These results comprehensively include a reduction in the number of reads and overhead for the competition procedures. Additionally, the bandwidth of the storage device affects the total execution time. We provide the results obtained by installing SATA SSD and NVMe SSD separately. In the figures, slight variations are present at some points, such as the 80% size of WCC and the 60% size of TC. This is because all experiments were individually conducted under non-deterministic parallelism as mentioned in Section 2.3.

When SATA SSD, with a relatively low bandwidth, is installed, the decreased number of reads directly influences the execution time. At the maximum, when the number of reads is reduced by 66% (70% the size of PR), the execution time is also reduced by 65%. On the other hand, with NVMe SSD, which provides much higher bandwidth, the disparity in execution time remains up to 28%. If the read requests from the graph workloads are too demanding, the bandwidth of SATA SSD is quickly satu-

rated, whereas NVMe SSD can offset this to some extent with its higher bandwidth. In our experiments, the PR algorithm is greatly affected by the bandwidth of the storage device because the sequential read dominates the entire I/O workload, as the graph engine contiguously accesses pages in vertex ID order.

In conclusion, GAC reduces the read volume by providing a higher hit ratio, resulting in comparable or improved performance, compared to existing implementation. On average, in all environments tested, GAC reduced the read volume by 14% and improved the performance by 11%. This was realized by the lightweight implementation and adaptability of the caching policy to the graph workload.

Chapter 5

Related work

Existing studies on out-of-core graph engines mainly optimize the access order to graph data with their partitioning and scheduling techniques. GraphChi [12] introduces the parallel-sliding-window model to load a partition of the destination vertices and a fraction of the source vertices from other partitions together. This minimizes redundant access within the superstep of the algorithm. X-stream [21] and GridGraph [22] perform the graph algorithms in edge order, hence the edge list pages are contiguously accessed. To prepare the irregular access to the vertex state data, a scatter-shuffle-gather model and a 2D grid partitioning technique are introduced, respectively. FlashGraph [6] supports a merging technique that coalesces the I/O requests to adjacent pages. GraFBoost [11] introduced a sort-reduce scheme with FPGA, reducing the data transmission for vertex state.

The above methods successfully streamline the out-of-core graph engines in a different way. However, their optimization techniques are realized in non-orthogonal design space, thereby a graph engine cannot enjoy their advantages at once. On the other hand, GAC has real-time adaptability to graph workloads. Therefore, it can display the

effectiveness on the top of different engines and under the different algorithm, dataset, and configurations.

Chapter 6

Conclusion

We have revealed the lack of effectiveness in an existing caching scheme for the out-of-core graph systems. In our analysis, existing static scheme showed a large variation in cache effectiveness according to the environmental factors such as graph algorithms, cache size, and dataset, etc. To handle this shortcoming, we proposed a graph-adaptive caching (GAC) by coordinating the competition procedures between the two policies, CLOCK and LIFO. It successfully reflects both the regularity and locality of graph workloads. In addition, we adjust the competition window based on fine-grained grouping and voter-centric competition techniques to realize our scheme with negligible overhead.

We applied our methodology to a recent out-of-core graph engine. To verify robustness against various conditions, we present the results with two types of graph data, four algorithms, and varying cache sizes. By way of experimentation, we prove that the proposed adaptive policy offers better effectiveness of cache and performance for out-of-core graph processing than a static policy.

Bibliography

[1] J. W. Lichtman, H. Pfister, and N. Shavit, "The big data challenges of connectomics," *Nat. Neurosci.*, vol. 17, no. 11, pp. 1448–1454, 2014.

[2] Statica, "Facebook monthly active users.", (accessed: 2020-12-10).

[3] S. Jiang, F. Chen, and X. Zhang, "Clock-pro: An effective improvement of the clock replacement.," in *USENIX Annual Technical Conference, General Track*, pp. 323–336, 2005.

[4] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs," in *Presented as part of the 10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*, pp. 17–30, 2012.

[5] J. Wang, Z. Liu, S. Ma, N. Ntarmos, and P. Triantafillou, "Gc: A graph caching system for subgraph/supergraph queries," *Proceedings of the VLDB Endowment*, vol. 11, no. 12, pp. 2022–2025, 2018.

[6] D. Zheng, D. Mhembere, R. Burns, J. Vogelstein, C. E. Priebe, and A. S. Szalay, "Flashgraph: Processing billion-node graphs on an array of commodity ssds," in *13th {USENIX} Conference on File and Storage Technologies ({FAST} 15)*, pp. 45–58, 2015.

[7] E. Lee, J. Kim, K. Lim, S. H. Noh, and J. Seo, "Pre-select static caching and neighborhood ordering for bfs-like algorithms on disk-based graph engines," in *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*, pp. 459–474.

[8] D. Zheng, R. Burns, and A. S. Szalay, "Toward millions of file system iops on low-cost, commodity hardware," in *SC'13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pp. 1–12, IEEE, 2013.

[9] Y. Low, J. E. Gonzalez, A. Kyrola, D. Bickson, C. E. Guestrin, and J. Hellerstein, "Graphlab: A new framework for parallel machine learning," *arXiv preprint arXiv:1408.2041*, 2014.

[10] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pp. 135–146, 2010.

[11] S.-W. Jun, A. Wright, S. Zhang, S. Xu, *et al.*, "Grafboost: Using accelerated flash storage for external graph analytics," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pp. 411–424, IEEE.

[12] A. Kyrola, G. Blelloch, and C. Guestrin, "Graphchi: Large-scale graph computation on just a {PC}," in *Presented as part of the 10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*, pp. 31–46, 2012.

[13] H. Liu and H. H. Huang, "Graphene: Fine-grained {IO} management for graph computing," in *15th {USENIX} Conference on File and Storage Technologies ({FAST} 17)*, pp. 285–300, 2017.

[14] H. Kwak, C. Lee, H. Park, and S. Moon, "What is twitter, a social network or a news media?," in *Proceedings of the 19th international conference on World wide web*, pp. 591–600, 2010.

[15] J. Wang, N. Ntarmos, and P. Triantafillou, "Graphcache: a caching system for graph queries," *International Conference on Extending Database Technology*, 2017.

[16] R. L. Bocchino Jr, V. S. Adve, S. V. Adve, and M. Snir, "Parallel programming must be deterministic by default," in *Proceedings of the First USENIX Conference on Hot Topics in Parallelism*, pp. 4–4, 2009.

[17] L. A. Belady, "A study of replacement algorithms for a virtual-storage computer," *IBM Systems journal*, vol. 5, no. 2, pp. 78–101, 1966.

[18] J. Chen and Y. Saad, "Dense subgraph extraction with application to community detection," *IEEE Trans. on Knowl. Data Eng.*, vol. 24, no. 7, pp. 1216–1230, 2010.

[19] M. D. Hill and A. J. Smith, "Evaluating associativity in cpu caches," *IEEE Trans. Comput.*, vol. 38, no. 12, pp. 1612–1630, 1989.

[20] J. Yang and J. Leskovec, "Defining and evaluating network communities based on ground-truth," *Knowl. Inf. Syst.*, vol. 42, no. 1, pp. 181–213, 2015.

[21] A. Roy, I. Mihailovic, and W. Zwaenepoel, "X-stream: Edge-centric graph processing using streaming partitions," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pp. 472–488, 2013.

[22] X. Zhu, W. Han, and W. Chen, "Gridgraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning," in *2015 {USENIX} Annual Technical Conference ({USENIX}{ATC} 15)*, pp. 375–386, 2015.

초록

그래프 엔진은 다양한 그래프 알고리즘과 데이터 하에서 효율적인 처리 능력을 보장하는 적응력을 갖춰야 한다. 외장형 그래프 엔진은 특히 위계적 메모리 구조를 활용하므로, 적응력 있는 캐시의 사용이 메모리 사용의 효용성을 유지하는 데에 결정적이다. 캐시 정책은 미래의 메모리 접근 워크로드에 대한 예측을 바탕으로, 사용될 것 같은 데이터를 상위 계층의 메모리에 보관한다. 하지만, 그래프 워크로드는 그래프 데이터, 알고리즘, 그리고 다양한 환경변수에 따라 메모리 접근 패턴이 변화하는 복잡성을 갖고있다. 따라서, 고정적 캐시 정책을 사용하는 것으로는 워크로드의 변화에 적절하게 대응하기 힘들다. 이 논문에서 우리는 그래프 적응력 있는 캐시 스킴을 제안하여, 변화하는 워크로드 하에서 일관성 있는 효용성을 제공하고자 한다. 우리의 캐시 스킴은 실시간 워크로드의 변화에 적응할 수 있는 캐시 정책을 채택한다. 그 변화를 감지하기 위해, 우리는 그래프 워크로드에 나타나는 대표적인 두 개의 상반된 속성—지역성 및 규칙성—사이의 경쟁 알고리즘을 도입했다. 뿐만 아니라, 경쟁 과정을 운영하는 오버헤드를 줄이기 위해 접근 범위를 조절하는 기법들을 결합한다. 우리가 제안하는 캐시 스킴은 다른 종류의 그래프 엔진에 접목되어, 메모리 사용의 효율을 높이는 데 기여한다. 실험 결과에서 이 스킴은 기존의 스킴과 비교했을 때 그래프 처리의 성능을 최대 65%까지 향상시키는 것으로 드러났다.

주요어: out-of-core graph processing, adaptive page replacement, memory access pattern, page cache optimization

학번: 2019-29953