공학석사 학위논문

Efficient Cache Simulation under Optimal Replacement on Large-scale Traces

대용량 트레이스에서의 최적 페이지 교체 알고리즘 향상 기법

2023 년 2 월

서울대학교 대학원
컴퓨터공학부
한 형 석

공학석사 학위논문

Efficient Cache Simulation under Optimal Replacement on Large-scale Traces

대용량 트레이스에서의 최적 페이지 교체 알고리즘 향상 기법

2023 년 2 월

서울대학교 대학원
컴퓨터공학부
한 형 석

Efficient Cache Simulation under Optimal Replacement on Large-scale Traces

대용량 트레이스에서의 최적 페이지 교체 알고리즘 향상 기법

지도교수 염 헌 영

이 논문을 공학석사 학위논문으로 제출함

2022 년 11 월

서울대학교 대학원

컴퓨터공학부

한 형 석

한 형 석의 공학석사 학위논문을 인준함

2023 년 1 월

위 원 장	엄 현 상	(인)
부위원장	염 헌 영	(인)
위 원	전 병 곤	(인)

Abstract

As the memory footprint of modern software continues to grow, efficient memory management is essential. Accordingly, studies on memory management policies are being actively conducted. To verify the policies and find the upper-bound performance of them, most studies use optimal replacement algorithm (OPT). Unfortunately, the existing well-known scheme of simulating OPT takes a lot of time, especially, when the trace size is large. Thus, it cannot calculate the hit ratio of the huge traces in a reasonable time.

In this paper, we propose high-performance OPT simulation techniques to reduce the time complexity and execution time of large-scale trace-driven simulation. To do this, first, we devised a data structure consists of an array and queues called *AccessMap* which reads the trace and stores the reference times per page in ascending order. It allows calculating the reference time of the accessed page in a constant time. Second, we applied a min-max heap to organize a cache based on min-max reference times. The min-max heap enables searching for the page and selecting an eviction target page optimally in a constant time. Finally, we leverage a Link-Tree for simulating multiple cache sizes on a single run as the previous study can do. Our evaluation demonstrates that the proposed scheme reduces the simulation time by up to 5.4x in single-size cache simulation and up to 4.4x in multiple-size cache simulation than the existing stack algorithm based scheme.

Keywords: optimal replacement, OPT, stack algorithm, page replacement
Student Number: 2021-22208

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The hardware requirements of modern software are growing as diverse services generally apply heavy tasks such as machine learning (ML) and big data processing [1, 10, 45, 53]. Today, however, hardware vendors are focusing on improving power efficiency and consolidation rather than improving the performance of hardware due to physical limitations [41, 44]. Studies to utilize hardware with better software techniques have logically followed as a result of this trend, which has persisted over the past few years [26, 30, 33, 34, 39, 40, 47, 50, 51]. In particular, in the case of the memory management algorithms or techniques, they are commonly compared with the optimal replacement algorithm (OPT and MIN) [5], which represents the upper bound hit ratio, to determine the performance of the techniques [7, 23, 29, 33, 34].

OPT is an offline algorithm, which evicts the farthest referenced entry in the cache [5]. In general, we do not know which memory address the process will access in the future, so OPT cannot be used in practical but simulated. While selecting a victim, OPT gets the next reference time of each entry in

the cache by traversing the access trace. This procedure has a time complexity of $O(C * N)$, C and N denote the block count of cache and total access count of trace. The linear time complexity for getting the next reference time makes OPT simulation takes a lot of time in big caches and huge access traces, which is common in modern software.

For simulating OPT efficiently, Mattson et al. [32] showed that OPT can be simulated by the stack processing. By adopting the stack processing, calculating the hit ratio for the whole cache size is available with a single run of the stack simulation. But this technique still suffers from the linear complexity of getting the next reference time of the block and updating the stack.

Sugumar and Abraham [42] proposed techniques for OPT stack simulation that preprocess the next reference of the entries in the limited time window and fix the errors that occur by the unknown next reference time blocks, to avoid traversing the trace each time the block enters the top of the stack. Also, they addressed the overhead of linear searching and updating the stack by grouping technique. However, even if the stack update procedure is optimized, the preprocess procedure still has a linear complexity to the window size.

In this paper, we propose LT-OPT, a high-performance OPT simulator for large-scale traces. We apply two novel techniques for LT-OPT's efficient OPT simulation. First, we apply a new data structure for OPT, `AccessMap`, which is a data structure consisted of an array and queues. Each element in the hash table corresponds to a page, one by one, and has a queue containing the access times to that address in ascending order. This helps to improve calculating the next reference time of a block to a constant time.

In addition, we devise "Link-tree", breaking away from the format using the existing stack algorithm. Link-tree is consist of several partition trees, which are implemented in min-max heap [2], and the size of each partition tree is decided

to target cache sizes entered by the user. This enables simulating multiple cache sizes efficiently.

We implement the simulator from scratch with `C++` for equivalent comparisons with the existing technique and other comparators. Our scheme outperforms Sugumar's scheme by up to 5.4x and showed faster simulation time with widely-used online algorithms such as `CLOCK` and `ARC`.

The contributions of our work are as follows:

- We investigate the performance bottleneck of a well-known OPT simulator.
- We propose a high-performance OPT simulator for large-scale traces called LT-OPT. It can reduce the simulation time while providing the correct hit ratio.
- We demonstrate that LT-OPT shows better performance by up to 5.4x.

The existing technique is explained in Chapter 2, and our scheme is described in Chapter 3. Chapter 4 explains how we implemented LT-OPT. Chapter 5 evaluates our scheme with others using large-scale traces from real-world workloads. Chapter 6 and 7 describes related works and conclusion, respectively.

Chapter 2

Background

2.1 Optimal Replacement Algorithm

The optimal replacement algorithm, as known as `OPT` [32] or `MIN` [5], minimizes the fault by evicting the block which is referenced in the farthest future. Because the algorithm evicts the block that will not be referenced sooner than all other blocks in the cache, OPT is the logically optimal solution for the given size of the cache.

OPT requires future access information. We normally can't know which address will be referenced in the future, so it is an offline algorithm that can't be run in practice. Even though OPT is not able to use in real-world applications, lots of memory management studies [23, 33, 7, 29, 34] use OPT for evaluations to compare with the maximum performance. Also, some of the studies such as [20], attempted to achieve better performance through a routine that works similarly to OPT.

Algorithm 1 OPT stack update

Require: M ▷ Total distinct data block count

Require: $S[1 \dots M]$ ▷ The algorithm's stack

```
1:  function STACK_UPDATE(address, priority)
2:      i ← 2
3:      cur_block ← S[1]
4:      S[1] ← (address, priority)
5:      while cur_block.address ≠ address and i ≤ M do
6:          if cur_block.priority > S[i] then
7:              swap(cur_block, S[i])
8:          end if
9:          i ← i + 1
10:     end while
11:     S[i] ← cur_block
12:     return i
13: end function
```

2.2 OPT Simulation with Stack Algorithm

Stack algorithm is introduced by Mattson et al. [32] for efficient hit ratio simulation. In the stack algorithm, the stack's behavior is quite different from the stack that we normally know–the Last-In-First-Out(LIFO) behavior. We briefly describe the mechanism of the stack algorithm in the cache simulation.

Algorithm 1 shows how the OPT stack simulation processes. When the address is referenced, it starts to check whether the data block of the address exists from the top of the stack. If found the block in the stack, its hit depth is the distance from the top of the stack. Else, the hit depth is infinite. The

algorithm replaces the top of the stack with the block of the address and keeps the previous block (line 3). Here, we call the block has been replaced as B. Then, it checks the priority of the next block under the replaced block, and if the block's priority is lower than B's priority, swaps B and the block. If the block's priority is greater than B's, continue to the next block under the current block (line 5-10). This procedure continues until the current position reaches the hit depth. When reached hit depth, put B into the hit depth of the stack and return (line 11-12). If the hit depth is infinite, the procedure ends at the end of the stack. As Sugumar et al. [42] did, we refer to this procedure as a "stack update" in this paper.

During the stack simulation runs, it saves the hit depth each time the stack update is called. A sequence of hit depths can be obtained after the simulation is finished. Then, a hit ratio of any cache size can be calculated with this sequence. Suppose the given cache size is N. If the hit depth is smaller or equal to N, the access is a cache hit, or else, a cache miss.

The stack algorithm can only be applied to the policies that are free from Belady's anomaly [6]. A phenomenon in which the hit ratio decreases even though the cache size is increased is called Belady's anomaly. Free from Belady's anomaly means that the blocks which are hit in cache size N must be hit in cache size N+1. In other words, the priority of the block is not associated with the page frame number of the block. We refer to this property as 'Belady-free', and the policy that is Belady-free as 'Belady-free policy'. `OPT` and `LRU` are representative of Belady-free policies. In the stack simulation of OPT, priority is bigger when the next reference time is sooner.

2.3 Optimizing Techniques for OPT Stack Simulation

Sugumar and Abraham [42] pointed out that OPT simulation using stack algorithm is too slow because it has reverse passes to the trace data for getting the time of the next reference during execution. As Sugumar et al. did, we refer to the block's next reference time as the block's TNR (Time Next Reference). For addressing this problem, they proposed "lookahead" technique. In their scheme, a communication buffer, which is a tuple array of (address, TNR), is maintained during the simulation. Before processing the stack algorithm, it first reads certain amount of accesses from traces and pushes pairs of (address, unknown) into the communication buffer. Each time it pushes a pair into the buffer, it updates the previous pair's priority which value is according to the current time.

Because the lookahead procedure's read size is limited, the inserted block's priority may be still not decided even until pushed into the stack. To resolve this problem, the lookahead procedure assigns dummy priorities for unknowns to make them able to repair when true priority is available. Dummy priorities have lower priority than any other known priority blocks, and in the stack, early-entered unknown blocks have higher priority than all other later-entered unknown blocks. When the unknown block's true priority becomes known, the stack repair procedure is called. Let B be pair of (known address, true priority), D be a temporal variable that can store a pair, U be dummy priority before it became known and T be the position that the address block exists in the stack. From the top of the stack, the routine first finds the first unknown block that has a lower priority than U. It swaps B and that block, and stores the evicted block into D. Then, until T, all entries which have lower dummy priorities than D are swapped. When it reaches T, it inserts D into the stack's T-th position.

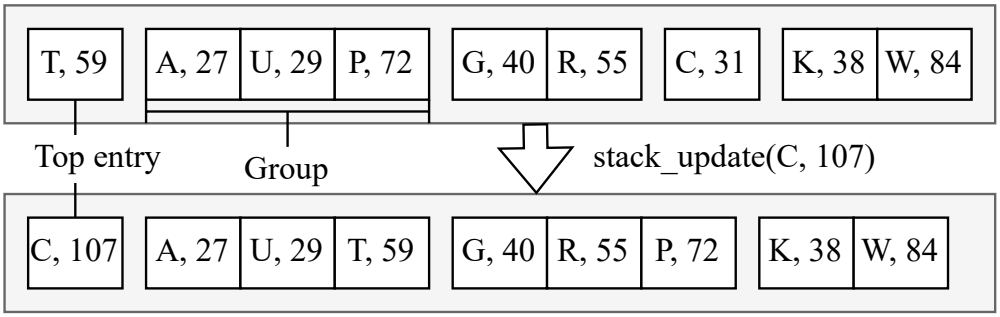

Figure 2.1 Stack simulation with grouping technique

By this procedure, the errors between the unknown block may be fixed.

With dummy priority, whenever the lookahead procedure pushes the pair, it checks whether the previous pair that has the unknown priority exists in the buffer. If exists, it updates the previous pair's TNR to the current time. If not exists, it means it's already pushed into the stack, so it calls the stack repair procedure with its true priority.

They also pointed out that searching the block in the stack is a significant overhead during the stack simulation. The grouping technique is their solution to this problem. They defined a group as a contiguous section of the stack that is sorted in descending order. According to this definition, since the block to be referenced earlier in the future exists at the top of the group, the hit always occurs only at the top of the group. Using this property, they improved the searching block operation to the linear time of the group. In addition, the stack update procedure which has to compare all blocks in the group is optimized in logarithmic time of the stack size by configuring each group as a tree.

2.4 Time Complexity and Space Overhead

The OPT simulation under the stack algorithm has a linear time complexity of N and H for getting the TNR of the entering block and the stack update respectively, where N and H denote the trace's access count and the hit depth. It is far better than Belady's OPT because the stack simulation can compute the hit ratio for any cache size in a single run. But, it still takes too much time to simulate huge traces where N and H are over the millions.

Sugumar and Abraham optimized the stack update procedure by the grouping technique, where the time complexity is the product of the number of groups and the number of blocks in the group. However, for updating the previous pair's TNR during the lookahead procedure, it has to traverse the communication buffer in sequential. Also, the effectiveness of the grouping technique may be reduced close to linear in the number of groups if the stack is divided into too many groups.

Algorithm 2 OPT stack processing

Require: N ▷ Total size of the trace

Require: $T[1 \dots N]$ ▷ The access records

Require: $S[1 \dots M]$ ▷ The algorithm's stack

Ensure: $H[1 \dots N]$ ▷ Hit depth sequence

```
1:  function MAIN
2:      for i ← 1 ... N do
3:          address ← T[i]
4:          next_access ← 0
5:          for j ← i ... N do
6:              if T[j] = address then
7:                  next_access ← j
8:                  break
9:              end if
10:         end for
11:         priority ← INT_MAX − next_access
12:         if S[1].address = address then
13:             H[i] ← 1
14:             S[1].priority ← priority
15:         else
16:             H[i] ← stack_update(address, priority)
17:         end if
18:     end for
19: end function
```

Chapter 3

Design

The hit ratio of any cache size can be determined via OPT simulation using the stack algorithm in a single run. On the opposite, it means that even if you only need a hit ratio for a small single cache size, you must perform the entire stack processing. Most studies only need the hit ratio of about ten cache sizes, so the whole stack processing can be inefficient.

Therefore, we got out of the stack algorithm and rethought it from the beginning. Basically, the OPT simulation proceeds in three phases for each reference of the trace: (1) getting the TNR of the referenced block, (2) searching the corresponding entry of the referenced block in the cache, and (3) if not found and eviction is needed, replace the cache entry of the biggest TNR with the entry of referenced block.

We propose LT-OPT, a high-performance OPT simulator for large-scale traces using Link-Tree. Figure 3.1 shows LT-OPT's overall structure and work-flow. When LT-OPT processes the reference, it first gets the current time of simulation and the TNR of the referenced block. Then it processes the Link-Tree

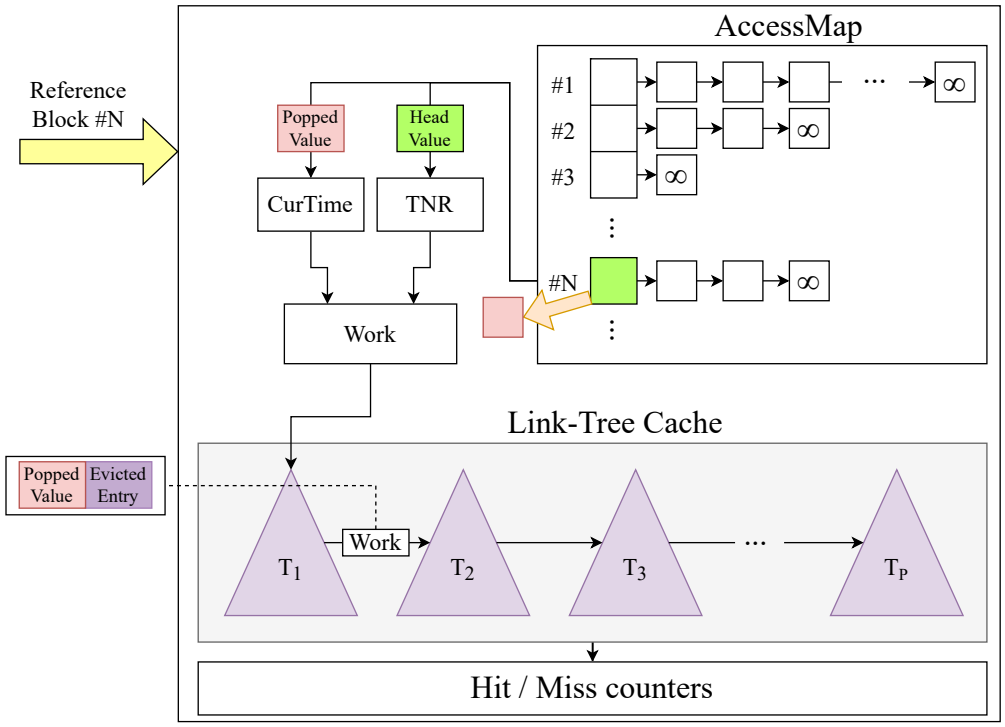

Figure 3.1 Overall structure of LT-OPT

operation with these values.

In chapter 3, we describe our optimization techniques applied in LT-OPT. Optimizing methods for each phase of OPT are discussed in sections 3.1 and 3.2 with an emphasis on a fast single run. We explain "Link-Tree" in section 3.3, which enables simulating multiple cache sizes on a single run.

The followings are the contributions of this paper:

- We optimized the normal operations of OPT simulation – searching whether the data block exists in the cache and getting the TNR of the block – to constant time complexity.
- We proposed techniques using Link-Tree, which can efficiently simulate

multiple cache sizes.

- Our scheme has lower space overhead than the existing technique.

3.1 AccessMap

As previously mentioned, the existing technique uses lookahead to decide the block's TNR before the stack processing, and it has a linear time complexity of the size of the communication buffer. To address the linear search of lookahead, we designed a new data structure named "AccessMap".

Figure 3.2 shows how AccessMap works. AccessMap consists of an array and queues. Each entry of the array is mapped to blocks one by one and has a single queue. Before the simulation starts, our simulator first initializes AccessMap by traversing the given trace once. It pushes the reference time of the block into the corresponding queue for each record of the trace. When the traversing trace is finished, lastly it pushes an infinite value to each queue. At the end of the initialization, as depicted in Figure 3.2-(a), each queue contains the access times of the block in ascending order.

After initializing AccessMap, the actual simulation starts from the beginning of the trace. Very first for each access, the simulator pulls the first entry from the queue corresponding to the referenced block. In Figure 3.2's case, block #A's queue is popped. Then, the value obtained when popping the queue is the current time during which the simulation is in progress, and the head of the queue is the TNR of the block. Next, the simulator searches the cache whether the corresponding cache entry to the reference block exists. If the cache entry exists, it updates the entry's TNR. Else, it replaces the entry of the biggest TNR from the cache with an entry of the referenced block.

The simulator gets the current time by popping the head, and the TNR of

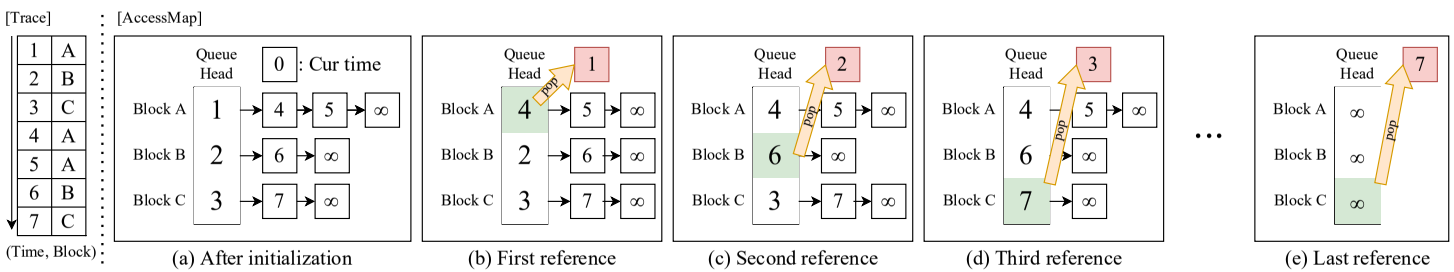

Figure 3.2 Visualization of AccessMap's operation

(a)-(e) shows the state of AccessMap from the initialization to the last access.

the block by looking in front of the corresponding queue. If the queue's head is infinite, it means the block will not be referenced anymore. The queue's `dequeue()` and `peek()` operation takes O(1) time, so getting the TNR operation can be optimized to constant from linear by applying the AccessMap. It also has a smaller space overhead with the communication buffer of Sugumar's lookahead, because queues only contain access times of blocks.

3.2 Fast One-Size Cache Simulation

For optimizing the search operation, we used a useful property of OPT simulation.

Theorem 1. *If the entry of the referenced block exists in the cache, it has the smallest TNR among all other entries in the cache.*

Proof. Let T be the current time. TNR is the entry's time of the next reference, so it can't be smaller than T. Therefore, before processing the T-th reference, the entries in the cache have TNR at least T. □

In accordance with the theorem 1, the simulator may determine if the current reference is a cache hit or cache miss by examining whether the minimum TNR of all items in the cache equals the current time. Additionally, this removes the requirement for the simulator's cache to maintain data indicating which cache item belongs to which block. All the simulator has to do in searching is check whether the entry of the current time exists in the cache.

We designed the cache of OPT simulator with a min-max heap [2] to best take advantage of theorem 1. Min-max heap has O(1) time complexity for obtaining minimum or maximum values in the heap. By adopting a min-max heap, the searching and the eviction target selection can be performed by getting minimum and maximum values, which are constant time complexity operations.

Also, the min-max heap's insertion and deletion take logarithmic time to the heap size, so replace operation takes O(log N) time.

Self-balancing trees such as the Red-Black tree [3] and AVL tree [16] can perform the same things and have equal time complexity to the min-max heap, but the min-max heap has advantages in performance and space overhead. Self-balancing trees perform pointer references, which can be random memory accesses, during tree operations such as balancing. Also, the nodes of self-balancing trees have to keep the pointer value of their child nodes. The min-max heap can find out the location of the children of a node with simple pointer arithmetic, so it has performance advantages by minimizing random memory access and also reducing space overhead.

Overall, for each reference, using the min-max heap size of N has O(log N) time complexity, whereas Sugumar et al.'s scheme has $O(Glog\frac{N}{G})$, N and G denote the size of the cache and the number of groups respectively.

3.3 Multiple Cache Size Simulation

Only improving single-size cache simulation is not enough, because most studies need tens of cache sizes for evaluating their caching scheme. LT-OPT also supports simulating multiple cache sizes on a single run, as the stack algorithm did. Our novel data structure, Link-Tree, can be used to simulate multiple cache sizes efficiently using the Belady-free property.

3.3.1 Link-Tree

For calculating the hit ratio in multiple cache sizes in a single run, like the stack algorithm, we introduce Link-Tree. We gave it this name since the tree operations are performed with the result of the previous tree.

Definition

The grouping technique proposed by Sugumar et al. inspired the basis for Link-Tree. Link-Tree is a collection of partition trees, and partition trees are implemented in min-max heaps. Link-Tree requires the array of cache size sorted in ascending order (hereinafter referred to as arr) and P as a parameter, where P indicates the number of partition trees. In the rest of our paper, we denote the minimum/maximum cache size in the input array as C_{min}, C_{max} and the count of distinct data blocks as M.

Each partition tree is assigned to one of levels 1 through P. In this paper, we define T_N to the partition tree of level N. Let S_N be a set of entries that exist in $arr[N-1]$ cache size. T_N contains entries excluding the entries exist in S_{N-1} among the entries exist in S_N at a specific time. That is, there are no duplicated entries among all partition trees, and the combination of all partition trees is a cache of size C_{max}. The Belady-free property makes this structure possible since it ensures that S_N is a subset of S_{N+1}.

Operation

Link-Tree works under 4 rules.

1. If T_N is full and has to insert an entry, it must pass an entry of max TNR to T_{N+1}.

2. If a block is referenced, it enters to T_1 regardless of its TNR. This is similar to how the stack algorithm always puts the just-referenced entry at the top entry.

3. For all partition trees except T_1, if the entry tries to enter the T_N and which has a bigger TNR than the max TNR of T_N, then T_N can just pass

the entry to T_{N+1} without insertion.

4. If the entry is crowded out from T_P, the entry is discarded.

Algorithm 3 shows the partition tree operation. Link-Tree performs partition tree operation from T_1 to T_P, sequentially and recursively. Partition tree operation requires the current time and E, which is the entry trying to insert, as parameters. It proceeds in three phases: Searching phase, updating phase, and calling phase. In the searching phase, compares the minimum TNR with the current time. If the minimum TNR equals the current time, the reference is a cache hit (line 2).

Then it continues to updating phase with the result of the searching phase. If it was a hit, increase the cache hit counter and update the cache entry of minimum TNR with E (line 3-5). The reference is missed in the current level if they are not equal. It means we need to look up whether the entry with the current time exists at the partition tree of the next level. In updating phase of miss, creates temporary space to keep the entry, and compares the maximum TNR of the partition tree with E. If the TNR of E is smaller than the maximum TNR, store the maximum TNR entry in the temporary space and swap the maximum TNR entry and E. Else, by rule 3, store E in the temporary space (line 7-13).

The calling phase is proceed only in case of a cache miss. In the calling phase, if a miss occurs in the partition tree that is not T_P, call the tree operation of the partition tree of the next level while passing parameters–the current time and the entry stored in the temporary space (line 14-15). Else, the current processing tree is T_P and it was a miss, then it is a cache miss in cache size C_{max}, so increase the miss counter (line 16-17).

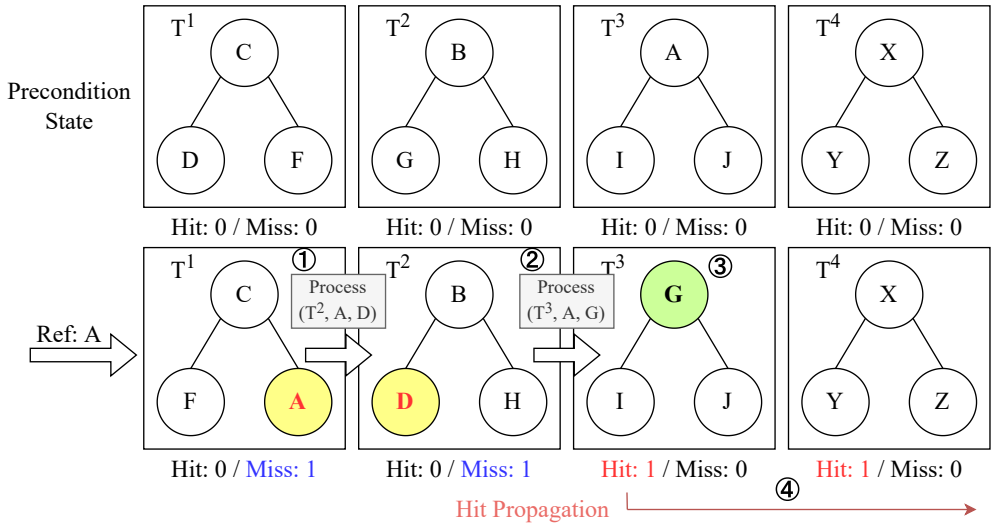

Figure 3.3 Example of multiple cache size simulation

3.3.2 Simulating with Link-Tree

For the multiple cache size simulation, LT-OPT takes an array of cache sizes sorted in ascending order to simulate as an input. Based on the input, LT-OPT determines the size of each partition tree and assigns a level to each tree. The first partition tree T_1 is created with the smallest cache size entered by the user(C_{min} == $arr[0]$). The rest of the partition trees are set to the size of *arr[N-2] - arr[N-1]*. For example, if the user entered {5, 10, 20} as an input, T_1 becomes size of 5 and T_2 and T_3 become size of 5 and 10 respectively.

LT-OPT maintains P hit/miss counters mapped to each partition tree. If a cache hit occurs in T_N, it is a cache hit in the bigger cache size of *arr[N-1]* by the Belady-free property. Therefore, by increasing all hit counters from T_N to T_P, the rest of the partition tree operation can be skipped. We refer to this tree operation skipping as 'hit propagation'.

Figure 3.3 shows how the Link-Tree process for multiple cache size simula-

tion. Link-Tree always starts at T_1 when processing references. (1) It searches for the entry of A in T_1 but it does not exist. So it increases the miss counter of T_1 swap the entry of maximum TNR and the entry of A. Then call the *process_part_tree*() function to search on the partition tree of the next level. (2) LT-OPT also searches for entry A in T_2 but there is no such entry. At this time, it compares the TNR of the entry crowded out from T_1 (entry D) with the maximum TNR entry of T_2 (entry G). If the latter has a bigger TNR, swap them and call the *process_part_tree*() with the swapped-out entry (the case of figure 3.3). Else, just call the next-level tree function with D without swapping with the maximum TNR entry of T_2. (3) LT-OPT searches in T_3 and found the entry of A. It increases the hit counter of T_3 and swaps entry A with the passed entry (G). (4) If there are more partition trees after the tree that hit occurred, skip all remaining tree operations because the increment in the hit tree represents it is all hit in all lower-level trees (hit propagation). In the case of figure 3.3, the hit counter of T_4 is incremented.

Algorithm 3 Partiton tree operation

Require: $T_1 \dots T_P$ ▷ Partition trees

Require: $HIT, MISS$ ▷ Hit/miss counter

1: **function** PROCESS_PART_TREE(T_N, CUR_TIME, E)
2: **if** $T_N.get\_min() == CUR\_TIME$ **then**
3: $HIT \leftarrow HIT + 1$
4: $T_N.pop\_min()$
5: $T_N.push(E)$
6: **else**
7: $max \leftarrow T_N.get\_max()$
8: $temp \leftarrow E$
9: **if** $E < max$ **then**
10: $temp \leftarrow max$
11: $T_N.pop\_max()$
12: $T_N.push(E)$
13: **end if**
14: **if** $T_N \neq T_P$ **then**
15: $process\_part\_tree(T_{N+1}, CUR\_TIME, temp)$
16: **else**
17: $MISS \leftarrow MISS + 1$
18: **end if**
19: **end if**
20: **end function**

Chapter 4

Implementation

We implemented LT-OPT and the cache simulators for other schemes or policies with `C++` from scratch. We used `gcc` compiler whose version is 8.4.0 which uses `c++14` in default. The optimization parameter is given to `-O3` for compiling the simulator.

We implemented the compare group including the stack algorithm applying the scheme of Sugumar et al., OSL [29], and online algorithms – LFU, CLOCK [11], LIRS [23], and ARC [33]. The simulator of Sugumar et al. is implemented in two versions. The difference is whether the group constituting the stack is implemented as a red-black tree or a min-max heap. Because `AccessMap` clearly optimizes TNR calculation than the lookahead process, it is applied to both versions of Sugumar.

We used a min-max heap implementation by itsjohncs [19]. We checked the correctness of the implementation by comparing the simulation results of several traces with the results of the Sugumar et al simulator.

Chapter 5

Evaluation

In this chapter, we present performance evaluations of LT-OPT. The goals of our evaluation are to show that LT-OPT has better performance than the scheme of Sugumar et al. and is even comparable to online algorithms having low overhead. We used 4-socket Intel Xeon Gold 6140 (2.3GHz, 18 cores per CPU) NUMA machine with 304 GB of total DRAM for evaluation. NUMA 0 node has 256 GB of DRAM and the other nodes have 16 GB each. We evaluated the performance by the execution time of the simulation. All simulators set their CPU affinity to the NUMA 0 node's first physical thread.

Table 5.1 describes the count of total references and distinct data blocks of traces used in the evaluation. Twitter_algorithm traces are extraction of block I/O operation of FlashX [54], a graph processing engine for large-scale graphs. We ran several algorithms(weakly connected components, diameter, pagerank2, cycle triangle, and localscan) with the Twitter Social Graph 2009 [27] on modified FlashX which saves data block reference requests in the log file. We also used traces from Microsoft MSR Cambridge block I/O trace [35]

Trace name	Total ref. count	# of distinct data blocks
twitter_wcc	9,439,516	6,096,612
twitter_diam	41,981,522	6,096,612
twitter_pr2	98,634,165	3,048,307
twitter_tc	1,432,095,672	6,096,612
twitter_ls	4,359,196,005	6,096,612
msr_proj	660,813,222	325,442,262
msr_prxy	576,411,570	3,824,259
msr_src1	818,690,304	63,853,879
msr_usr	745,221,453	272,785,468

Table 5.1 Trace information

for our evaluation. We chose `proj`, `prxy`, `src1`, and `usr` traces which have a large reference count.

5.1 Fast Cache Simulation

For showing the efficiency of our scheme, we evaluated LT-OPT for several situations in 5.1. First, we evaluated LT-OPT in the case of simulating one cache size, comparing it with Sugumar et al's technique. Next, we compare the performance of LT-OPT when simulated simultaneously for multiple cache sizes.

5.1.1 Single-Size Cache Simulation

We first evaluate our LT-OPT with the previous scheme. Figure 5.1 shows the runtime of single-size OPT simulations, normalized by the runtime of the simulator of Sugumar et al. Execution time of singe-size cache simulation time

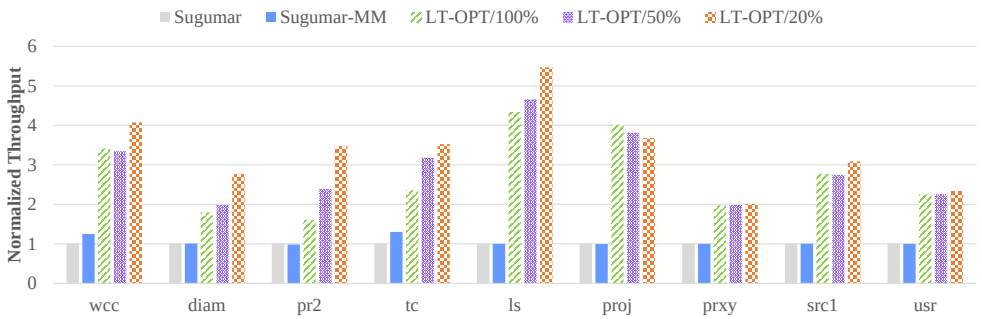

Figure 5.1 Single-size cache simulation throughput comparison

normalized to the scheme of Sugumar et al. LT-OPT/N% indicates when P=1 and the maximum cache size is set to 100%, 50%, and 20% of the trace's memory footprint. Sugumar-MM is implementing the scheme of Sugumar et al. using a min-max heap. The N of LT-OPT/N% means the target cache size is N% of the total memory usage of the workload.

For all traces, LT-OPT always outperforms Sugumar and showed better performance up to 5.46x in the bigger reference count. We can also observe that limiting the maximum size of the cache has extra performance improvement in Twitter workloads. In contrast, MSR workloads showed minimal performance improvement with small cache sizes, and `proj` even showed degradation of performance.

The lower degree of performance improvement by smaller cache size is due to two factors. First, reference to data blocks that are not referenced before – we refer to this as 'cold reference' – dominates the trace. At the cold reference, if the cache is not full, LT-OPT does not need to evict entries from the cache; it only needs to perform an insert operation. However, if the cache is full, cache eviction is required to store the block just referenced, and at this time, delete and insert operations must be performed together. That is, some

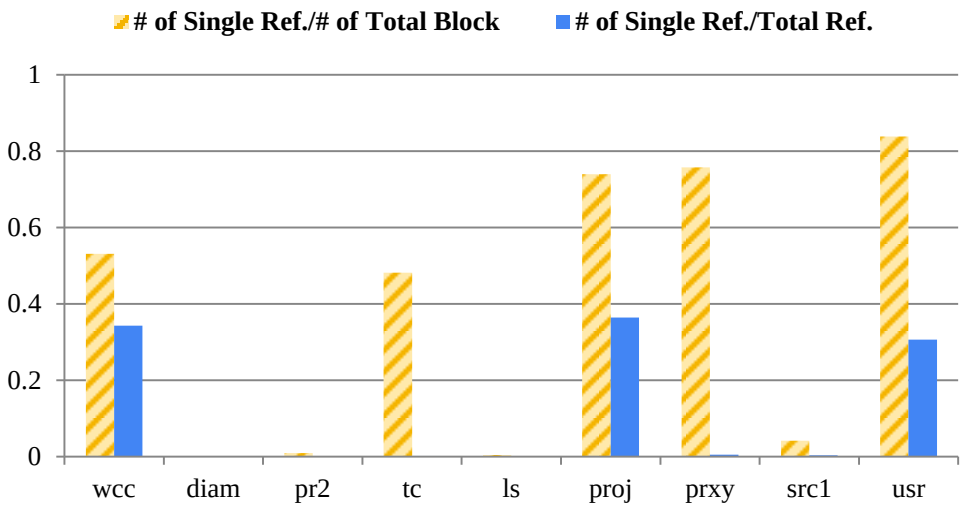

Figure 5.2 The ratio of the number of single-referenced blocks

of the references that only needed an insert operation in the 100% size cache additionally require a delete operation in the 50% size cache. Therefore, the performance can be degraded when the overhead due to the additional eviction is greater than the performance gain due to the smaller operation. The yellow, hatch bar of figure 5.2 depicts the ratio of the number of blocks referenced only one time (single-referenced blocks) to the number of total distinct data blocks. `wcc`, `proj`, and `usr` are cold-reference-dominated traces, and we can see performance degradation in `proj`. But `wcc` showed performance improvement in LT-OPT/20%, because the gain of the smaller operation becomes greater than the additional insert operation overhead.

Second, even if the cold references do not occupy a large part of the trace, it is difficult to see a significant effect on performance if cold blocks are referenced relatively slowly. The effect of limiting the cache size comes from making the overhead of insert/delete operations as large as $log(cache\_size)$. However, if the

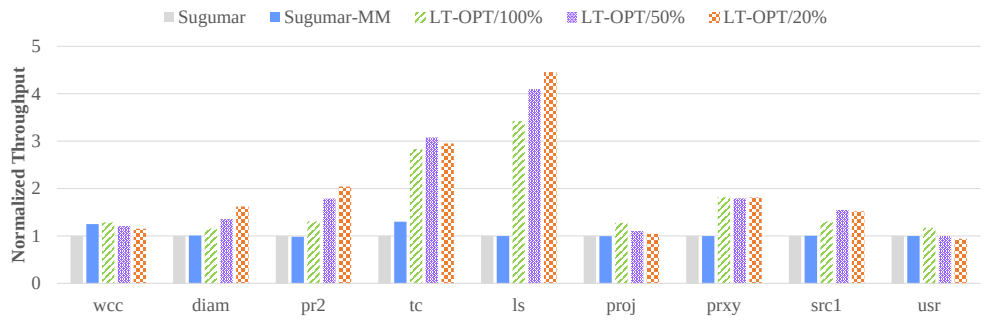

Figure 5.3 Multiple-size cache simulation throughput comparison

cache is not full, the operations take as much time as the log of the number of entries being used in the cache. It is for this reason that MSR traces show overall lower performance than Twitter traces. Since Twitter is a graph algorithm workload, it references a lot of data relatively more quickly than MSR, which recorded block I/O requests of users on the cluster for a week.

Also, we can see that the min-max heap has performance benefits by comparing Sugumar and Sugumar-MM on twitter_wcc and twitter_tc. This is because the min-max heap minimizes random memory accesses during the simulation.

5.1.2 Multiple-Size Cache Simulation

Next, we evaluate the multiple-size cache simulation performance of LT-OPT. We set the maximum/minimum size of the target cache size to 100%/10%, 50%/30%, 20%/10% and denote each to LT-OPT/100%, LT-OPT/50%, and LT-OPT/20%. Each LT-OPT set its number of partition trees to 10 (P=10), so each of them simulates the hit ratio for 10 intervals between the minimum and maximum cache size.

Figure 5.3 shows the throughput normalized to Sugumar. LT-OPT showed

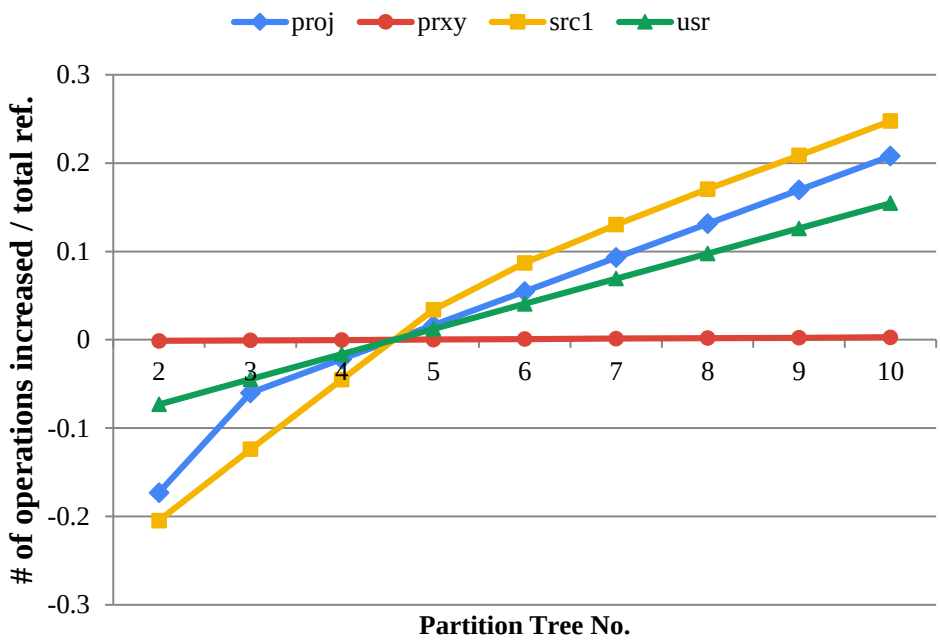

Figure 5.4 The difference in the number of processed operations between the partition trees of LT-OPT/100% and LT-OPT/50%

at least a similar performance to Sugumar, and outperformed up to 4.45x for traces that have large reference counts (`tc`, `ls`). We could determine LT-OPT will show more significant performance improvement in larger traces.

In detail, there is performance degradation between LT-OPT/100% and LT-OPT/50% for `proj` and `usr`. This happens because simulating a smaller cache will result in more insert operations; the performance benefit of the smaller cache size was offset by the insert overhead. For example, suppose there is a cache that can store a total of 10 entries. Then inserting 10 entries into this cache takes 10 operations. But if this is done with 2 partition trees, operations must be performed 10 times in the first partition tree and 5 times in the second partition

tree, for a total of 15 operations. As a result, the effect of maintaining a small cache is mitigated by the increase in the overall number of operations carried out. A cold miss that impacts all partition trees with entries at the moment of occurrence highlights the issue this causes. In other words, the "domination of cold reference" indicated in single-size cache simulation makes the performance even worse in multi-size cache simulation. LT-OPT gains efficiency by keeping the partition tree as small as possible and reducing operations on the partition tree through hit propagation. However, because it has no impact, performance in `proj` and `usr` declines to a level comparable to Sugumar.

We can observe this problem in figure 5.4. It shows how many more operations were processed into the partition trees in LT-OPT/50% than LT-OPT/100%. In `proj` and `usr`, partition trees of LT-OPT/50% processed fewer operations up to T_4, and starts to process more operations below T_5. The reduction in processed operations up to T_4 is because the first partition tree at LT-OPT/50% manages 30% of the distinct data blocks alone, while at LT-OPT/100% it manages splits from T_1 to T_3. But on average, each partition tree except T_1 handled 5% and 4% more operations, so taken together, the total number of processed operations increased by 41.8% and 36.6% of the total reference count of the trace. That is, as can be seen from figure 6, all references except for cold misses are hits, but the increase in the total number of operations due to cold misses degrades the performance.

Meanwhile, `src1` has a higher increase rate of operations than `proj` and `usr` but showed performance improvement. This is because the increment of processed operations due to cold misses does not take many portions of total operations. Since most references of `src1` is not cold reference, operations occur in many groups in Sugumar as well. Therefore, the performance improvement in `src1` is the result of showing the structural efficiency of LT-OPT.

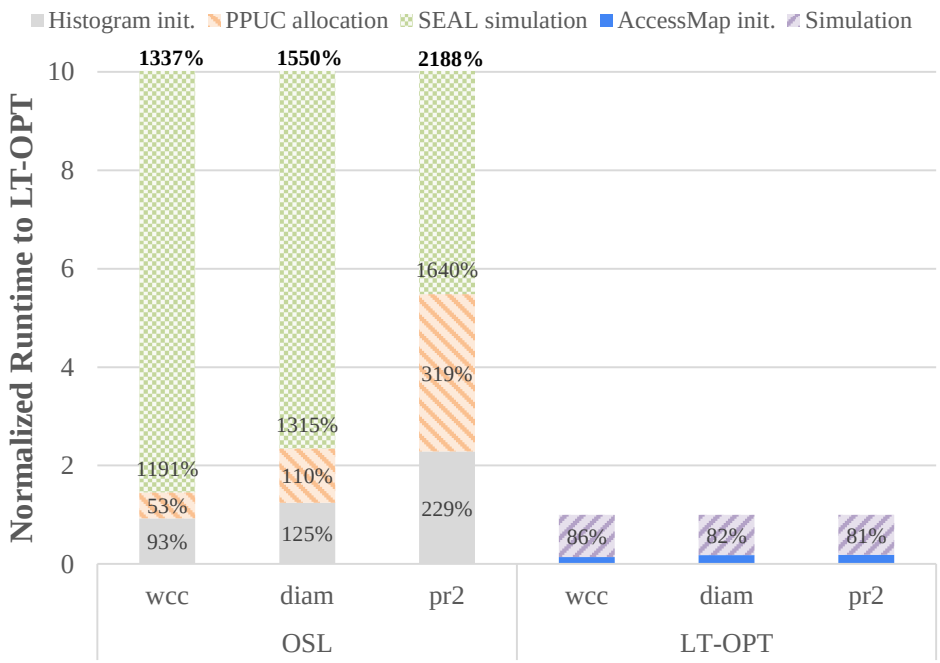

Figure 5.5 Runtime analysis of OSL and LT-OPT

5.2 Evaluate with Other Algorithms

In this section, we compare runtime with OSL [29] and some online algorithms that are widely used–CLOCK, LFU, LIRS, and ARC. For all comparisons, LT-OPT determined the runtime as the sum of AccessMap initialization time and actual simulation time. The runtime of OSL is determined by the sum of the histogram/PPUC initialization and PPUC allocation time plus the actual simulation time using SEAL. Lastly, for all online algorithms, the runtime is calculated as the sum of the simulation time for determining all of the target cache sizes of LT-OPT/100%.

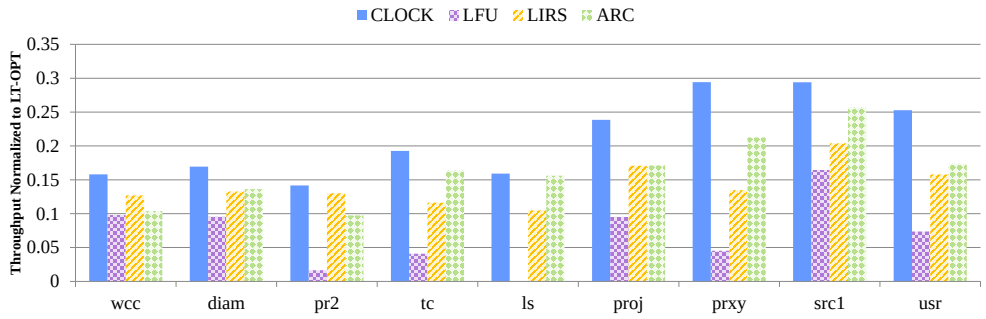

Figure 5.6 Throughput of online algorithms normalized to the throughput of LT-OPT/100%

5.2.1 vs. OSL

To compare LT-OPT with OSL, we set OSL's target average cache size set equal to LT-OPT/20%. In addition, OSL's lease assignment has been implemented with optimization applied to reuse the PPUC allocation of the previous cache size. Lastly, we set the timeout to 24 hours to run OSL. Figure 5.5 shows detailed runtime of both OSL and LT-OPT. Due to the significant runtime difference, only up to 1000% is indicated in the graph. The PPUC initialization process is contained in the histogram init. The value written inside the bar graph is the value normalized from the runtime of the relevant part to the entire runtime of LT-OPT. The bolded label at the top of the OSL bar indicates the OSL runtime versus LT-OPT. Only the three traces that finished within the timeout period, `wcc`, `diam`, and `pr2` were compared.

As a result, LT-OPT outperformed OSL by at least 13x. OSL exceeded the execution time of LT-OPT only by generating histogram and calculating PPUC even for relatively small traces such as diam. In addition, OSL's SEAL applied optimization by managing leases in buckets, but was greatly affected by deallocation overhead for entries in the bucket of L=1 for every reference.

On the other hand, LT-OPT can initialize the necessary data structures simply by reading the trace once, and can perform efficient simulation through caches only with the required size. Through comparison of the results with OSL, it can be seen that LT-OPT makes OPT simulation efficient compared to offline algorithms.

5.2.2 vs. Online Algorithms

Finally, we evaluate our LT-OPT with online algorithms. Figure 5.6 shows the performance comparison with LT-OPT and online algorithms. A higher value indicates higher throughput. For all traces, LT-OPT outperforms all online algorithms by at least 3.4x. `list` [12] of C++ STL was used as buffers for CLOCK, ARC, and LIRS, and data blocks and iterators were mapped using STL `map` [13] to determine whether they existed in the cache and used for search. LFU is implemented using `set` [14] of C++ STL, and it tracks the reference count of the entries currently present in the cache. Overall, the algorithm that took the longest to simulate is LFU, because it uses logarithm time to increase the block counter for every reference. As a result, LT-OPT can simulate faster than the algorithm actually used online by effectively optimizing the searching and tree operation.

Chapter 6

Related Works

6.1 Cache Management Policies

For the past decades, lots of caching policies are proposed for efficient cache use to reduce the performance penalty of cache misses. Minimizing such penalties leads to smaller reference time of applications, so adopting an efficient cache management policy is critical for higher performance in both computer architecture and operating systems.

There are basically two kinds of reference locality: temporal locality and spatial locality. Temporal locality means that specific data is referenced again within a short period of time, and spatial locality is the tendency of data in the vicinity of the referenced data to be referenced. Based on this, it is possible to estimate whether data will be referenced in the future through past reference trends. Least-Frequently Used (LFU) reflects these localities by evicting the least referenced data by storing the number of times the data is referenced. In addition, Least-Recently Used(LRU) assumes that the recently referenced data

will be needed in the future, and the most recently referenced data evict the oldest data. CLOCK [11] is an approximation version of LRU, and instead of all pages having a referenced time, a second chance is given through a reference bit to keep the recently referenced page.

Because CLOCK performed well in the portion of workloads, a variant of CLOCK was applied in many OSes including Linux and Windows [17]. However, LRU-based policies such as CLOCK show serious performance degradation in reference patterns where the assumptions of LRU do not fit. For example, when a reference pattern such as a memory scan occurs, CLOCK loses useful information it previously had. This is because the LRU only considers *data recency* and not *data frequency*.

To address this problem, recent studies have suggested a method of adaptively managing the cache according to the progress of workloads. LRFU [28] combines LRU and LFU and chooses to give more weight based on history. LRU-K [36] supplemented LRU's lack of data frequency by approximating LFU based on recent K references. 2Q [24] is the same as LRU-2, but highly available in practice with constant time complexity. LIRS [23] manages the cache by dividing it into Low Inter-reference Recency (LIR) and High Inter-reference Recency (HIR) partitions based on the reuse distance suggested by Mattson et al. CLOCK-Pro [22] is an approximation of LIRS, and the Linux buffer cache replacement implementation uses a combination of LRU and CLOCK-Pro [31]. ARC [33] manages the cache with hot/cold LRU buffers and adjusts the size of each buffer based on the metadata of recently evicted pages to operate adaptively to the workload. In particular, ARC is adopted by many systems as a "golden standard" for storage caching [48]. In addition to the studies mentioned so far, studies for improving cache performance are being actively conducted [8, 21, 25, 37, 46, 52]. By quickly calculating the baseline of performance, our

work is beneficial for further research on cache management policies.

6.2 Variable-size Cache

While fixed-size cache limits the maximum size of the cache, variable-size cache limits the average usage of the cache. They have a different mechanism for managing their buffer. The fixed-size cache "reactively" manages its space–it makes free space by eviction when the cache is full. In contrast, the variable-size cache is "prescriptive", which means that it manages its space by allocation.

The concept of the variable-size cache is discovered by Denning [15]. Denning defined the working set model, which tracks the recent reference behavior of a process. The variable-size cache adapts its size when the working set of the program is changed. Even though the working set theory is proposed about a half-century ago, it is still positioned as one of the foundations of the memory management system of modern operating systems.

The optimal for the variable-size cache, VMIN, is given by Prieve and Fabry [38]. Li et al. proposed OSL(Optimal Steady-state Lease)[29], which tried to mimic VMIN with statistical clairvoyance and variable-size cache. They used the cache lease concept for allocating cache blocks, which is initially used in the distributed file caching system [18].

Because fixed-size and variable-size caches have different constraints and criteria, comparison between them is generally not appropriate. But as Li et al. did, we can still use OPT as a baseline of performance to determine whether the scheme based on variable size cache successfully adapts to the working set.

6.3 Miss Ratio Curves(MRCs)

Cache allocation needs to be improved in order to maximize cache performance. MRCs are a tool used to determine the dynamic memory demand of workload, and are useful in both software cache and hardware cache. MRCs are the curves of miss rate per memory size, and the result obtained by OPT simulation for a range of specific memory sizes is the same as the MRC. Using this fact, studies such as Hawkeye [20] and Pacman [9] adopted the scheme of OPT for optimizing cache performance. Hawkeye learned and used a model that mimics OPT's decision based on past information. Pacman used OPT's stack distance to optimize caching in loop-based code.

There are studies, though, that do not employ the OPT scheme. RapidMRC [43] improves L2 cache performance through hardware support in the form of Performance Monitoring Units (PMUs). SHARDS [49] approximated MRC through spatial sampling at representative locations. Talus [4] mitigated the MRC cliff with LRU cache partitioning to make the performance improvement according to the cache size consistent. SLIDE [48] obtained transparent cliff removal effect through scaled-down simulation.

As a result, OPT is not practical because it requires future reference information, but it can be used as a baseline in research to improve cache performance using MRC. Additionally, our scheme will be advantageous to future studies that use the scheme of OPT for cache optimization.

Chapter 7

Conclusion

This paper has proposed LT-OPT, the high-performance OPT simulator for large-scale traces. LT-OPT optimized searching the block and eviction target selection operations in constant time complexity by adopting AccessMap and min-max heap. LT-OPT also minimized memory overhead by advancing the useful fact that OPT simulation can be processed with only TNR. We also presented a Link-Tree capable of simulating multiple cache sizes concurrently, which enables more efficient simulation than stack processing simulation. Link-tree creates several partition trees and its structure reduces unnecessary memory usage according to the settings provided by the user. It addresses the problem of Sugumar et al's grouping technique in which there may be too many groups by fixing the number of partition trees. In comparison to the scheme of Sugumar et al., LT-OPT achieved up to 5.4x higher performance in single-size cache simulation and up to 4.4x higher performance in multiple-size cache simulation. We observed that LT-OPT has a more significant performance improvement on the bigger traces.

Bibliography

[1] Ahmed Abdelaziz, Mohamed Elhoseny, Ahmed S Salama, and AM Riad. A machine learning model for improving healthcare services on cloud computing environment. *Measurement*, 119:117–128, 2018.

[2] Michael D Atkinson, J-R Sack, Nicola Santoro, and Thomas Strothotte. Min-max heaps and generalized priority queues. *Communications of the ACM*, 29(10):996–1000, 1986.

[3] Rudolf Bayer. Symmetric binary B-trees: Data structure and maintenance algorithms. *Acta informatica*, 1(4):290–306, 1972.

[4] Nathan Beckmann and Daniel Sanchez. Talus: A simple way to remove cliffs in cache performance. In *Proceedings of the IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2015.

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

[6] Laszlo A Belady, Robert A Nelson, and Gerald S Shedler. An anomaly in space-time characteristics of certain programs running in a paging machine. *Communications of the ACM*, 12(6):349–353, 1969.

[7] Daniel S Berger, Ramesh K Sitaraman, and Mor Harchol-Balter. {AdaptSize}: Orchestrating the hot object memory cache in a content delivery network. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017.

[8] Aaron Blankstein, Siddhartha Sen, and Michael J Freedman. Hyperbolic caching: Flexible caching for web applications. In *Proceedings of the 2017 USENIX Annual Technical Conference (ATC)*, 2017.

[9] Jacob Brock, Xiaoming Gu, Bin Bao, and Chen Ding. Pacman: Program-assisted cache management. *ACM SIGPLAN Notices*, 48(11):39–50, 2013.

[10] Rubén Casado and Muhammad Younas. Emerging trends and technologies in big data processing. *Concurrency and Computation: Practice and Experience*, 27(8):2078–2091, 2015.

[11] Fernando J Corbato. A paging experiment with the multics system. Technical report, MASSACHUSETTS INST OF TECH CAMBRIDGE PROJECT MAC, 1968.

[12] cppreference.com. std::list, 2022.

[13] cppreference.com. std::map, 2022.

[14] cppreference.com. std::set, 2022.

[15] Peter J Denning. The working set model for program behavior. *Communications of the ACM*, 11(5):323–333, 1968.

[16] Caxton C Foster. A generalization of AVL trees. *Communications of the ACM*, 16(8):513–517, 1973.

[17] Mark B Friedman. Windows NT page replacement policies. In *Proceedings of the International Computer Measurement Group Conference (CMG)*, volume 99, pages 234–244, 1999.

[18] Cary Gray and David Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. *ACM SIGOPS Operating Systems Review*, 23(5):202–210, 1989.

[19] itsjohncs. minmaxheap-cpp, 2016.

[20] Akanksha Jain and Calvin Lin. Back to the future: Leveraging Belady's algorithm for improved cache replacement. In *Proceedings of the ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 78–89. IEEE, 2016.

[21] Aamer Jaleel, Kevin B Theobald, Simon C Steely Jr, and Joel Emer. High performance cache replacement using re-reference interval prediction (RRIP). *ACM SIGARCH computer architecture news*, 38(3):60–71, 2010.

[22] Song Jiang, Feng Chen, and Xiaodong Zhang. CLOCK-Pro: An effective improvement of the CLOCK replacement. In *Proceedings of the 2005 USENIX Annual Technical Conference (ATC)*, 2005.

[23] Song Jiang and Xiaodong Zhang. LIRS: An efficient low inter-reference recency set replacement policy to improve buffer cache performance. *ACM SIGMETRICS Perform. Eval. Rev.*, 30(1):31–42, 2002.

[24] Theodore Johnson, Dennis Shasha, et al. 2Q: a low overhead high performance bu er management replacement algorithm. In *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB)*. Citeseer, 1994.

[25] Hyojun Kim and Seongjun Ahn. BPLRU: A buffer management scheme for improving random writes in flash storage. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST)*, volume 8, pages 1–14, 2008.

[26] Mariam Kiran, Peter Murphy, Inder Monga, Jon Dugan, and Sartaj Singh Baveja. Lambda architecture for cost-effective batch and speed big data processing. In *Proceedings of the IEEE International Conference on Big Data (Big Data)*. IEEE, 2015.

[27] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is Twitter, a social network or a news media? In *Proceedings of the 19th international conference on World wide web*, 2010.

[28] Donghee Lee, Jongmoo Choi, Jong-Hun Kim, Sam H Noh, Sang Lyul Min, Yookun Cho, and Chong Sang Kim. LRFU: A spectrum of policies that subsumes the least recently used and least frequently used policies. *IEEE transactions on Computers*, 50(12):1352–1361, 2001.

[29] Pengcheng Li, Colin Pronovost, William Wilson, Benjamin Tait, Jie Zhou, Chen Ding, and John Criswell. Beating opt with statistical clairvoyance and variable size caching. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.

[30] Gangmuk Lim, Jeongseob Ahn, Wencong Xiao, Youngjin Kwon, and Myeongjae Jeon. Zico: Efficient GPU memory sharing for concurrent DNN training. In *Proceedings of the 2021 USENIX Annual Technical Conference (ATC)*, 2021.

[31] LinuxMM. Pagereplacementdesign, 2017.

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

[33] Nimrod Megiddo and Dharmendra S Modha. ARC: A Self-Tuning, low overhead replacement cache. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST)*, 2003.

[34] Jason Mohoney, Roger Waleffe, Henry Xu, Theodoros Rekatsinas, and Shivaram Venkataraman. Marius: Learning massive graph embeddings on a single machine. In *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 533–549, 2021.

[35] Dushyanth Narayanan, Austin Donnelly, and Antony Rowstron. Write offloading: Practical power management for enterprise storage. *ACM Transactions on Storage (TOS)*, 4(3):1–23, 2008.

[36] Elizabeth J O'neil, Patrick E O'neil, and Gerhard Weikum. The LRU-K page replacement algorithm for database disk buffering. *ACM SIGMOD Record*, 22(2):297–306, 1993.

[37] Seon-yeong Park, Dawoon Jung, Jeong-uk Kang, Jin-soo Kim, and Joonwon Lee. CFLRU: a replacement algorithm for flash memory. In *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, 2006.

[38] Barton G Prieve and Robert S. Fabry. VMIN—an optimal variable-space page replacement algorithm. *Communications of the ACM*, 19(5):295–297, 1976.

[39] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimizations toward training trillion parameter models. In *Pro-*

ceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC). IEEE, 2020.

[40] Liana V. Rodriguez, Farzana Yusuf, Steven Lyons, Eysler Paz, Raju Rangaswami, Jason Liu, Ming Zhao, and Giri Narasimhan. Learning cache replacement with CACHEUS. In *Proceedings of the 19th USENIX Conference on File and Storage Technologies (FAST)*, 2021.

[41] John Shalf. The future of computing beyond Moore's law. *Philosophical Transactions of the Royal Society A*, 378(2166):20190061, 2020.

[42] Rabin A. Sugumar and Santosh G. Abraham. Efficient simulation of caches under optimal replacement with applications to miss characterization. New York, NY, USA, 1993.

[43] David K Tam, Reza Azimi, Livio B Soares, and Michael Stumm. RapidMRC: approximating L2 miss rate curves on commodity systems for online optimizations. *ACM Sigplan Notices*, 44(3):121–132, 2009.

[44] Thomas N. Theis and H.-S. Philip Wong. The end of Moore's law: A new beginning for information technology. *Computing in Science & Engineering*, 19(2):41–50, 2017.

[45] Zaib Ullah, Fadi Al-Turjman, Leonardo Mostarda, and Roberto Gagliardi. Applications of artificial intelligence and machine learning in smart cities. *Computer Communications*, 154:313–323, 2020.

[46] Giuseppe Vietri, Liana V Rodriguez, Wendy A Martinez, Steven Lyons, Jason Liu, Raju Rangaswami, Ming Zhao, and Giri Narasimhan. Driving cache replacement with ML-based LeCaR. In *Proceedings of the 10th*

USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage), 2018.

[47] Álvaro Villalba, Josep Lluís Berral, and David Carrera. Constant-time sliding window framework with reduced memory footprint and efficient bulk evictions. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 30(3):486–500, 2019.

[48] Carl Waldspurger, Trausti Saemundsson, Irfan Ahmad, and Nohhyun Park. Cache modeling and optimization using miniature simulations. In *Proceedings of the 2017 USENIX Annual Technical Conference (ATC)*, 2017.

[49] Carl A Waldspurger, Nohhyun Park, Alexander Garthwaite, and Irfan Ahmad. Efficient MRC construction with SHARDS. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST)*, 2015.

[50] Chien-Yao Wang, Hong-Yuan Mark Liao, Yueh-Hua Wu, Ping-Yang Chen, Jun-Wei Hsieh, and I-Hau Yeh. CSPNet: A new backbone that can enhance learning capability of CNN. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, 2020.

[51] Pinghui Wang, Yiyan Qi, Yu Sun, Xiangliang Zhang, Jing Tao, and Xiaohong Guan. Approximately counting triangles in large graph streams including edge duplicates with a fixed memory usage. 2017.

[52] Kan Wu, Zhihan Guo, Guanzhou Hu, Kaiwei Tu, Ramnatthan Alagappan, Rathijit Sen, Kwanghyun Park, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. The storage hierarchy is not a hierarchy: Optimizing caching on modern storage devices with orthus. In *Proceedings of the 19th USENIX Conference on File and Storage Technologies (FAST)*, 2021.

[53] Xiang Yan, Xinyu Liu, and Xilei Zhao. Using machine learning for direct demand modeling of ridesourcing services in chicago. *Journal of Transport Geography*, 83:102661, 2020.

[54] Da Zheng, Disa Mhembere, Randal Burns, Joshua Vogelstein, Carey E Priebe, and Alexander S Szalay. FlashGraph: Processing Billion-Node graphs on an array of commodity SSDs. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST)*, 2015.

초록

현대 소프트웨어의 메모리 요구량이 점점 늘어남에 따라, 효율적인 메모리 관리를 위한 연구들이 활발히 이루어지고 있다. 최적 페이지 교체 알고리즘(이하 OPT)는 가장 나중에 참조되는 페이지를 교체하는 오프라인 알고리즘으로, 페이지 교체 정책에 대한 연구에서 상한 성능으로써 사용된다. 하지만 기존의 OPT 시뮬레이션 기법은 현대 소프트웨어의 대용량 메모리 트레이스를 합리적인 시간 안에 시뮬레이션하지 못하고 있다.

본 논문에서는 대용량 트레이스를 효율적으로 처리할 수 있는 새로운 시뮬레이션 기법을 제안한다. 기존 기법의 다음 참조 시간 계산의 시간 복잡도를 개선하기 위해, 배열과 큐로 구성된 AccessMap을 새롭게 적용한다. AccessMap의 각 큐에는 해당하는 페이지가 접근하는 시간이 오름차순으로 정리되어 다음 참조 시간을 상수 시간에 계산할 수 있도록 했다. 또한 최댓값과 최소값을 상수 시간에 수행할 수 있는 Min-Max 힙을 적용해, 페이지 검색과 교체할 페이지 선정에 걸리는 시간을 최적화한다. 마지막으로, 여러 캐시 크기에서의 적중률을 한 번의 실행으로 효율적으로 계산할 수 있는 Link-Tree를 새롭게 고안해 적용한다.

실험 결과, 제안한 기법은 기존 기법 대비 단일 크기의 캐시 시뮬레이션에서는 최대 약 5.4배 높은 처리량을 보였으며, 여러 크기의 캐시를 동시에 시뮬레이션 할 때는 최대 약 4.4배 높은 처리량을 보였다. 또한 트레이스의 크기가 커짐에 따라 더 높은 성능 향상을 확인할 수 있었다.

주요어: 최적 페이지 교체, OPT, 스택 알고리즘, 메모리 관리

학번: 2021-22208