

國立臺灣大學電機資訊學院電機工程學系

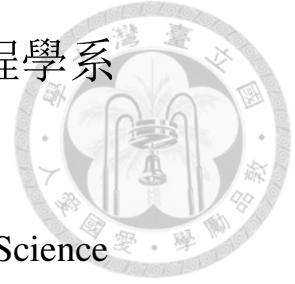
碩士論文

Department of Electrical Engineering

College of Electrical Engineering and Computer Science

National Taiwan University

Master Thesis



基於快取特性之記憶體式資料庫批次更新
Cache-Aware Batch Update for In-Memory Databases

張庭綱

Ting-Kang Chang

指導教授：陳銘憲 博士

Advisor: Ming-Syan Chen, Ph.D.

中華民國 106 年 7 月

July, 2017

國立臺灣大學碩士學位論文
口試委員會審定書

基於快取特性之記憶體式資料庫批次更新
Cache-Aware Batch Update for In-Memory Databases

本論文係張庭綱君（學號 R04921045）在國立臺灣大學電機工程學系完成之碩士學位論文，於民國 106 年 7 月 20 日承下列考試委員審查通過及口試及格，特此證明。

口試委員：

陳銘憲

（簽名）

（指導教授）

修正承

黃仁璋

陳昭倫

系主任

劉志文

（簽名）



Acknowledgments

First of all, I would like to express my sincere gratitude to my advisor, Prof. Ming-Syan Chen, who introduced me to the opportunity to study on this interesting topic, and gave me many helpful suggestions when I was lost in pointless thoughts.

Also, this work is supported by ITRI (Industrial Technology Research Institute) in Taiwan. I appreciate Director Tzi-Cker Chiueh for his insightful suggestions, which helped me improve the work, and led me to break through the bottlenecks. In addition, I am thankful for the colleagues of ITRI: Chen-Ting Chang, Yu-Lun Chen, and Hung-Hsuan Lin, who constructed and maintained the experiment environment, and gave valuable advice during the weekly meetings.

I am grateful for my fellow mates in Network Database Laboratory, who helped me improve this work and finish the thesis. It's my pleasure to learn and work with them.

Finally, I must thank my family for providing me spiritual and financial support throughout my years of study. I could not have come this far without their encouragement and support.



中文摘要

低延遲、高流通量的記憶體式資料庫管理系統(DBMS)近年來因為硬體的發展，受到了研究與應用領域越來越多的關注。更甚者，有許多因應未來使用非揮發性隨機存取記憶體(NVRAM)之記憶體式儲存系統的研究也被提出。然而這些研究皆沒有對於在低局部性、密集的更新作業負載下的處理器快取利用率進行討論。此類負載容易造成低度的快取利用率，導致不理想的整體效能。我們設計了一個基於快取特性之批次更新架構藉以提升在此類負載下之效能。藉由將更新請求暫存於快取中，此系統可將數個於不同時間到達之相近請求聚合並在同一批次內更新，以避免對記憶體不必要的重複存取，進而減少快取未命中(Cache miss)並提升整體流通量。實驗結果顯示本論文提出的快取模型相對於未考量快取特性的參考模型，能節省最高達75%的末級快取(Last level cache)未命中數，以及達到最大65%的速度提升。



Abstract

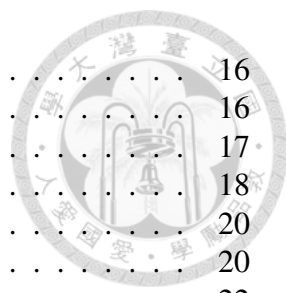
Low-latency, high throughput in-memory DBMSs have been attracting attention in research and application increasingly in recent years, thanks to hardware evolutions. Furthermore, there are many works proposed to address issues in the upcoming NVRAM era for durable in-memory storage. However, very few of them were focused on cache utilization for low-locality update-intensive workloads, which usually lead to poor cache utilization and undesirable performance. We design a cache-aware batch update model to improve cache efficiency for such workloads. By buffering update requests in cache, the system can aggregate spatially close requests arriving at distant time into one batched update, and avoid unnecessary data re-fetch from memory, thus reducing read latency and improve overall throughput. The experiments show that our proposed cache-aware model can save up to 75% last-level-cache misses and achieve up to 1.65x speedup over a cache-oblivious reference model.



Contents

口試委員會審定書	i
Acknowledgments	ii
中文摘要	iii
Abstract	iv
1 Introduction	1
2 Preliminaries	3
2.1 Batch Update	3
2.2 Cache-Oblivious Data Update	3
2.2.1 Distributed-Input Configuration	3
2.2.2 Cache interference	4
2.2.3 Memory bucket miss rate	4
3 Cache-Aware Batch Update	6
3.1 In-cache Batch Update	7
3.1.1 Batch aggregation ratio	7
3.2 Relative Per-Request Misses	8
3.3 Decoupled Threading	8
3.3.1 Lock-free threading	9
3.3.2 Flushing request queues	10
4 Experiments & Discussions	11
4.1 Experiment Environment	11
4.1.1 NUMA effect	11
4.1.2 NUCA effect	12
4.2 Experiment Settings	12
4.2.1 Data & request model	12
4.2.2 Searching data structure	12
4.2.3 General factors settings	13
4.3 Modeling the Cost per Input Request	13
4.4 Workload Formation	14
4.4.1 Uniformly random	14
4.4.2 Clustered random	14
4.5 Cache Partition using Coloring	15
4.5.1 Side effect of CControl	15

4.6	Cache-Buckets Layout	16
4.7	Overhead of Cache-Aware Model	16
4.8	Changing memory Bucket Size	17
4.9	Changing Effective Memory Bucket Number	18
4.10	Relative LLC Miss Count	20
4.11	Summarization of All Factors	20
4.12	Improvement in Cost vs. Improvement in LLC-Misses	22
5	Conclusion	23
	Bibliography	24

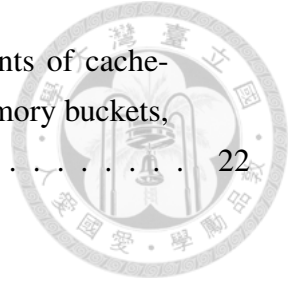




List of Figures

2.1	Overview of BOSC conception	3
2.2	The structure of cache-oblivious baseline model.	4
3.1	The structure of <i>foreground</i> tasks of cache-aware model.	7
3.2	An illustration of batch-update. The small squares form a cache-bucket, each represents a buffered request. A request belongs to the memory bucket with the same label.	8
3.3	The ratio of theoretical LLC-miss-rate between cache-aware and baseline model. X-axis is thousands memory bucket $m/1000$. The other parameters are from one of our experiment.	9
3.4	The structure of decoupled foreground-background cache-aware model. .	10
4.1	An illustration of request processing cost.	14
4.2	Plot Equation 3.2, M_c versus c . Let $m/nc = 4$	15
4.3	The average base cost for handling a request per worker thread, in terms of CPU cycles. The value is calculated by dividing the time elapsed by number of request handled per thread, and transforming the unit by multiplying the master CPU frequency.	17
4.4	The average cost of handling a request, in unit of cycles. The effective # of memory buckets is 400,000 and the record size is set as 16Bytes. . . .	18
4.5	The relative throughput of cache-aware model over baseline, versus # of effective memory buckets.	19
4.6	The relative LLC miss count of cache-aware model over baseline, versus # of effective memory buckets.	19
4.7	The experiment results of relative LLC misses of cache-aware model, compared with theoretical values in Equation 3.4.	20
4.8	The bar charts show the relative throughput values of cache-aware model over baseline. The X-axis is effective # of memory buckets, multiplying 1000.	21

4.9 The bar charts show the relative measured LLC-miss counts of cache-aware model over baseline. The X-axis is effective # of memory buckets, multiplying 1000. 22





List of Tables

4.1	The specifications of experiment platform	11
4.2	The general settings of various factors. There are total 108 cases.	13
4.3	The average throughput of three cache-bucket organizations. $r=16B$, $m=400k$ $a=512B$	16
4.4	The average LLC-misses of three cache-bucket organizations. $r=16B$, $m=400k$ $a=512B$	17
4.5	The fitted linear model of per-request cost cycles versus per-request LLC- misses. Each entry represents a model from data of 12 distinct (m, a) combinations.	22



Chapter 1

Introduction

As the main memory evolves to be larger and cheaper, it becomes possible to use byte-addressable, random access memory instead of hard disks as the main storage for database management systems (DBMSs), to achieve low latency service for modern applications who have stricter performance requirements. Much research has been devoted to develop such in-memory database management systems [24]. Unlike traditional disk-based DBMSs, who usually care about I/O efficiency, in-memory DBMSs consider relatively more on other performance factors, such as CPU efficiency [4, 17] or concurrency control [4, 7, 22]. In addition, CPU cache utilization is also an increasingly important factor for in-memory DBMSs [18, 24]. For example, many cache-conscious index structures have been proposed [12, 13]. Although in-memory DBMSs can benefit much from low latency and high bandwidth of memory, it is still necessary to store durable data or logs into persistent disks for recovery, if fault tolerance is required [24]. To avoid expensive I/O for disk logging which potentially results in performance bottleneck, many approaches such as group commit have been provided [4].

Furthermore, with the expectation of upcoming non-volatile random access memory (NVRAM) era for durable in-memory database management, several NVRAM-based DBMS designs [5, 9, 15], as well as improved index structures for NVRAM [3, 23] have been proposed recently. While all these approaches spend more efforts on cache-awareness than traditional disk-based systems, few of them consider the cache efficiency with workload locality concern. For low-locality update-intensive workloads, which are commonly produced by index updates in data de-duplication, user generated content management and on-line transaction processing applications [20], could result in poor cache utilization and few in-cache reuse due to limited cache capacity, and finally lead to undesirable throughput. In order to alleviate the problem of poor cache utilization for low-locality workloads in memory resident database, we propose a cache-centric update request handling technique inspired by *Batching mOdifications with Sequential Commit*

(BOSC) [20], which is proposed to improve the performance of disk-based storage system for low-locality update-intensive workloads.

For in-memory database, an index update requires loading a region of memory into the cache for CPU to operate searching. If the workload has poor locality, that is, arriving update requests spread across a wide range within a short period, the total size of memory regions required for handling these requests will not fit in the last-level cache. This results in competition of cache capacity, and the loaded cache lines could be evicted from cache before being referenced again, while a re-reference is expected for updating data in the same memory region. In order to reduce these unwanted evictions, we design a data structure called *cache-buckets*, which buffers incoming update requests. By buffering update requests, we can aggregate the updates that require references to the same memory region into a batch and apply them at a time. This helps achieving higher temporal locality of memory reads. Thus the system needs only single time of memory region access to solve multiple requests, reducing the unnecessary evictions and re-fetching in comparison with cache-oblivious model.

The major contributions of this thesis are:

1. Propose a cache-centric batch update model using a specialized request buffering data structure called *cache-buckets*, which reduces cache misses and improves overall throughput for low-locality update-intensive workloads.
2. Provide an analytic model which can roughly predict the relative last-level-cache miss counts between proposed cache-aware and baseline cache-oblivious model with locality factor and average read size per request factor.
3. Perform experiments on implemented cache-aware model and baseline model with varieties of factor configurations. The experiment results show that cache-aware model can save up to 75% last-level-cache misses and gain up to 65% increase in throughput in comparison with cache-oblivious baseline.

The remainder of this thesis is organized as follows. In Chapter 2, we introduce the concept of batch update, the reference baseline model and some existing techniques applied for proposed model. Chapter 3 describes our cache-aware batch update model in detail. We provide experiment results and discussions in Chapter 4. Finally, we conclude the thesis with Chapter 5.



Chapter 2

Preliminaries

In this chapter, we'd like to introduce the basic concept about batch update for disk-based systems and our cache-oblivious baseline model for analysis with proposed cache-aware model.



Figure 2.1: Overview of BOSC conception

2.1 Batch Update

The *Batching mOdifications with Sequential Commit* (BOSC) [20] system maintains a set of *data disks* to store data and a set of in-memory disk update request queues. By buffering update requests in the memory, the system can aggregate several requests which refer to the same disk block into a batch, and complete them with single disk block read and write. Thus most expensive random disk access can be reduced.

2.2 Cache-Oblivious Data Update

We design a cache-oblivious update model as the baseline in our experiments.

2.2.1 Distributed-Input Configuration

We assume that each thread can access the input interface on its own simultaneously. Thus the baseline model is called *distributed-input* model. As shown in Figure 2.2, there are

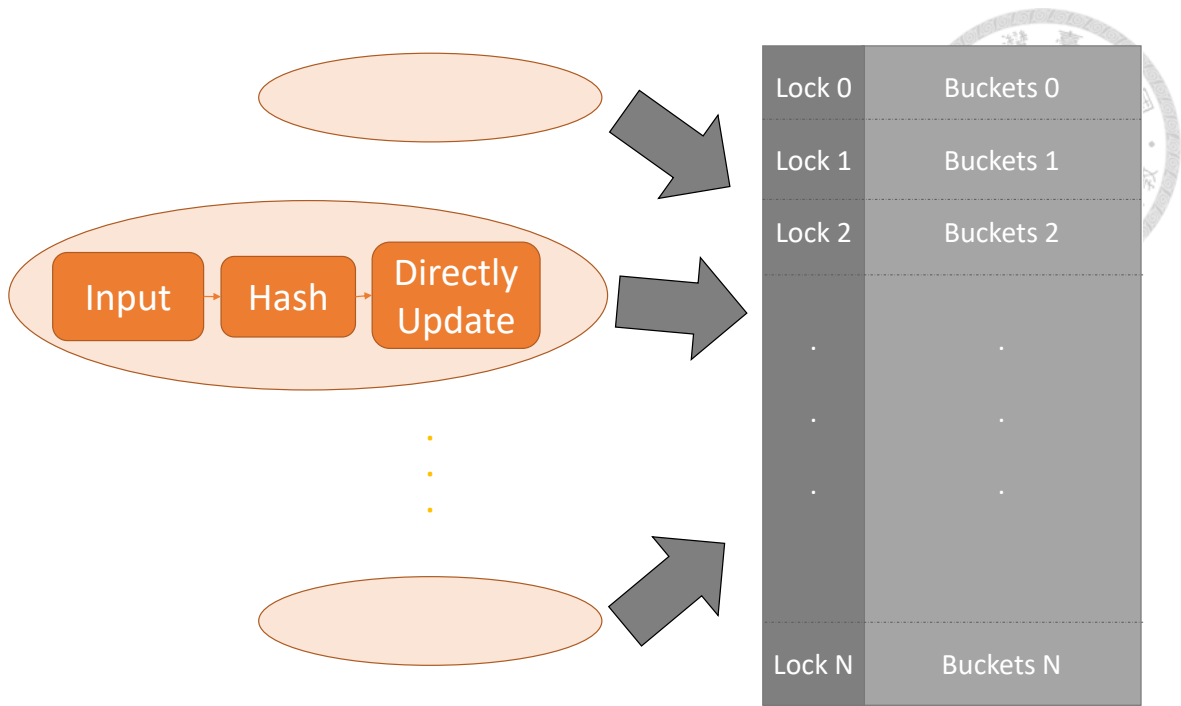


Figure 2.2: The structure of cache-oblivious baseline model.

multiple identical threads. Each thread keeps taking update requests from input, calculating the corresponding hash key from request record key and then directly applying each of them into its corresponding memory bucket. Since the memory buckets are not separated into exclusive regions, it is possible that a memory bucket is required by multiple threads at a time, so locks are necessary to preserve consistency.

2.2.2 Cache interference

On each update request, CPU must read data from the corresponding memory bucket, for searching the target position of the request. If required data present in CPU cache, update operations can start sooner and faster. If not, the CPU will fetch the data from memory to CPU cache, and will evict other data from the same cache set, if the cache set is full. Unfortunately, the capacity of cache is too small for every memory bucket to stay in, so evictions happen frequently. Thus, the probability of a bucket being in the cache when required is low, producing many memory accesses.

2.2.3 Memory bucket miss rate

For simplicity of theoretical analysis of cache misses, we assume that the number of cache misses is proportional to number of requests. Thus we can instead discuss *per-request* miss rate. Furthermore, we assume that the memory bucket is fetched into and evicted from cache atomically. Let the total available last-level-cache size be S , the number of

memory buckets be m , and the average required (that is, accessed) size for each update request be a . Then with uniformly random distributed access to each bucket, we have the probability that a memory bucket is still cached when it's going to be updated again. That is, the per-request cache hit-rate in long term is

$$H_b = \frac{S}{ma}, \quad (2.1)$$

assuming the cache usage of other data is negligible.

By subtracting 1 by Equation 2.1, we can also get the per-request cache miss rate as

$$M_b = 1 - H_b = 1 - \frac{S}{ma}. \quad (2.2)$$

We will compare the per-request miss rate of cache-oblivious model with that of our cache-aware model later in Chapter 3.



Chapter 3

Cache-Aware Batch Update

The key cause of poor bucket reuse rate for uniformly random access is the low temporal locality of bucket accesses. If the system buffers update requests for a period and applies spatially close requests at a time, the temporal locality of memory bucket accesses can be increased. That is, each time the system loads a memory bucket from memory into cache, multiple update requests are solved, and the rate of memory accesses can be reduced.

To achieve such target, we designed cache-aware batch update model, which is equipped with a cache-centric data structure named *cache-buckets*. As its name implies, there are a set of buffer buckets, each of whom can hold several requests in it. With the total size not larger than the size of last-level-cache (LLC), the cache-buckets are expected to stay in the cache most of the time, and thus the system can access them with cache speed. When a cache-bucket gets full, the system will flush all the buffered requests in the cache-bucket, by executing a batch update.

As shown in Figure 3.1, in addition to the main memory buckets, there is a set of in-cache-buckets. The system does not directly apply an input request to its corresponding memory bucket. It instead puts the request into its corresponding cache-bucket. We call taking input requests, calculate corresponding cache-bucket keys and buffer the requests into cache-buckets as *foreground* jobs. Unlike the main buckets, the cache-buckets are small enough and expected to stay in the cache, holding buffered input requests. Before a cache-bucket getting full, there is no need to read from memory. All the tasks can be done within the LLC, with high speed and low latency accesses on cached data.

When a cache-bucket is found full, the thread which is holding the cache-bucket's lock will flush the buffered requests and apply them to their corresponding memory buckets. We call popping out buffered requests from a cache-bucket, calculating corresponding memory bucket keys and finally applying them to the memory buckets as *background* tasks. In the following section, we will describe the flushing procedure in detail.

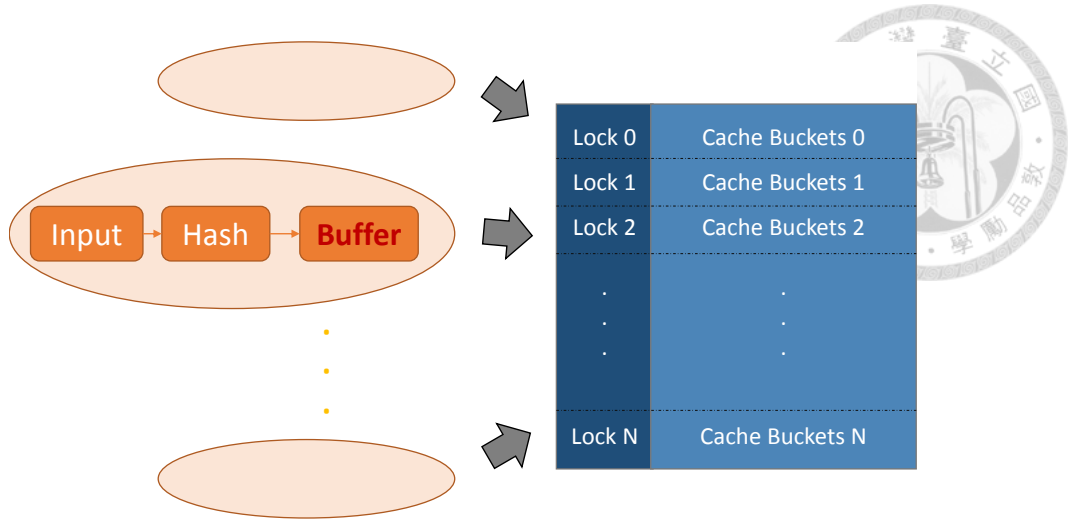


Figure 3.1: The structure of *foreground* tasks of cache-aware model.

3.1 In-cache Batch Update

Upon a cache-bucket being full, the system will manage a batch-update on it. Since each cache-bucket has been mapped to several memory buckets, one batch-update may requires accesses to multiple memory buckets. In Figure 3.2, the cache-bucket has 4 corresponding memory buckets. This implies that batch-update on it requires at most 4 memory bucket reads. In fact, there are chances that only 3 or less memory buckets are required in a single batch. As Figure 3.2 shows, the buffered requests belong to A, B, and D memory buckets. That is, in this batch, the system needs to access 3 memory buckets and can solve 8 requests. Then we have per-request bucket miss rate of $3/8$ in this case. We will discuss the average per-request bucket miss rate in the following section.

3.1.1 Batch aggregation ratio

In each flush of a cache-bucket, there are c requests to be solved. On the other hand, we have to load multiple memory buckets in the batch since requests in a cache-bucket could be mapped to m/n memory buckets, if there are n cache-buckets. Then the expected number of actually mapped memory buckets is

$$\rho = \frac{m}{n} \left(1 - \left(\frac{m-n}{m} \right)^c \right). \quad (3.1)$$

We can solve c request with only ρ memory bucket reads, resulting a *aggregation ratio* written as ρ/c .

Assuming each load produces a memory bucket miss, we have the per-request miss rate as

$$M_c = \frac{\rho}{c} = \frac{m}{nc} \left(1 - \left(\frac{m-n}{m} \right)^c \right). \quad (3.2)$$

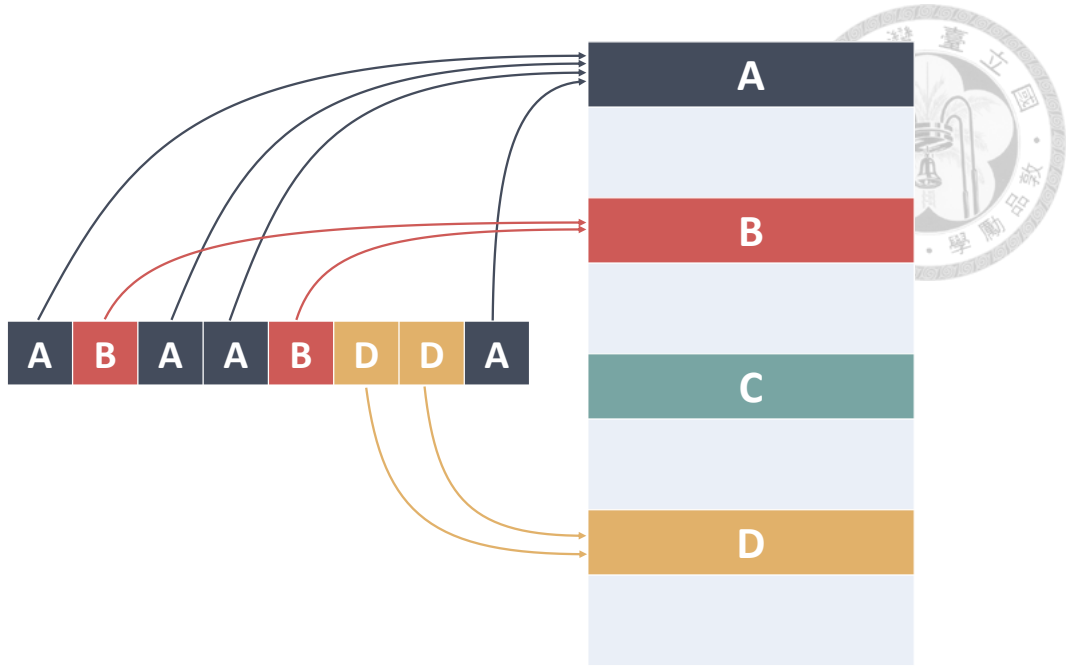


Figure 3.2: An illustration of batch-update. The small squares form a cache-bucket, each represents a buffered request. A request belongs to the memory bucket with the same label.

3.2 Relative Per-Request Misses

By assuming that the cache-buckets occupy almost the whole last-level-cache, we can rewrite the total LLC size S as

$$S = ncr, \quad (3.3)$$

where r is the size of each request.

Then the relative per-request miss count of cache-aware model over baseline can be obtained from Equation 2.2, 3.2 and 3.3:

$$M_r = \frac{M_c}{M_b} = \frac{\frac{m}{nc} \left(1 - \left(\frac{m-n}{m}\right)^c\right)}{1 - \frac{ncr}{ma}}. \quad (3.4)$$

We can plot the theoretical relative miss rate of cache-aware model over baseline. As Figure 3.3 shows, there is a knee point in the curve with position related to a . With few memory buckets (the left side of Figure 3.3), almost all the working set can fit in the CPU cache, so the baseline model has very low miss rate without cache-awareness and there is no room of improvement in cache miss rate.

3.3 Decoupled Threading

In addition to the foreground-background dual-mode identical threads model, we also propose a model which separates the request-handling tasks into two parts, as foreground

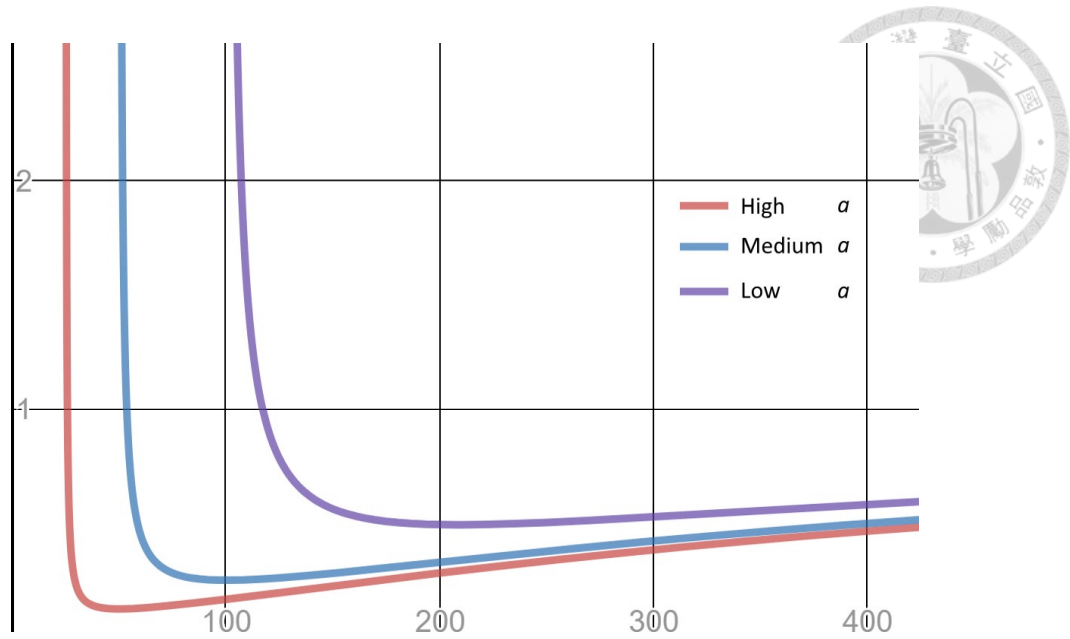


Figure 3.3: The ratio of theoretical LLC-miss-rate between cache-aware and baseline model. X-axis is thousands memory bucket $m/1000$. The other parameters are from one of our experiment.

tasks and background tasks, and assign foreground tasks and background tasks to different threads.

There is one or several threads, called *foreground thread(s)*, keeping accepting requests from input interface, and putting each request into its corresponding buffering cache-bucket. We call the other threads *background threads* or *flusher threads*. They keep flushing buffered input requests from cache-buckets into memory buckets. Since the cache-buckets are expected to stay in the cache most of the time, foreground thread can thus always run at cache speed.

However, due to limited size of current cache, the capacity of cache-buckets cannot be high enough to keep requests for a reasonably long request duplicate interval. So there is no in-place update for current cache-buckets model, and the requests remain unsolved until they are flushed from the cache-bucket. One possible way to solve this problem is to apply fault-tolerance techniques such as write-ahead-logging between cache and NVRAM data blocks, and let read requests always take the latest version. This could be an issue of future study.

3.3.1 Lock-free threading

One advantage of decoupled threading is there can be no locks. If there is only one thread in foreground, it can access any cache-bucket at any time, without conflict concern. For background threads, we can divide the cache-buckets into several exclusive and complete

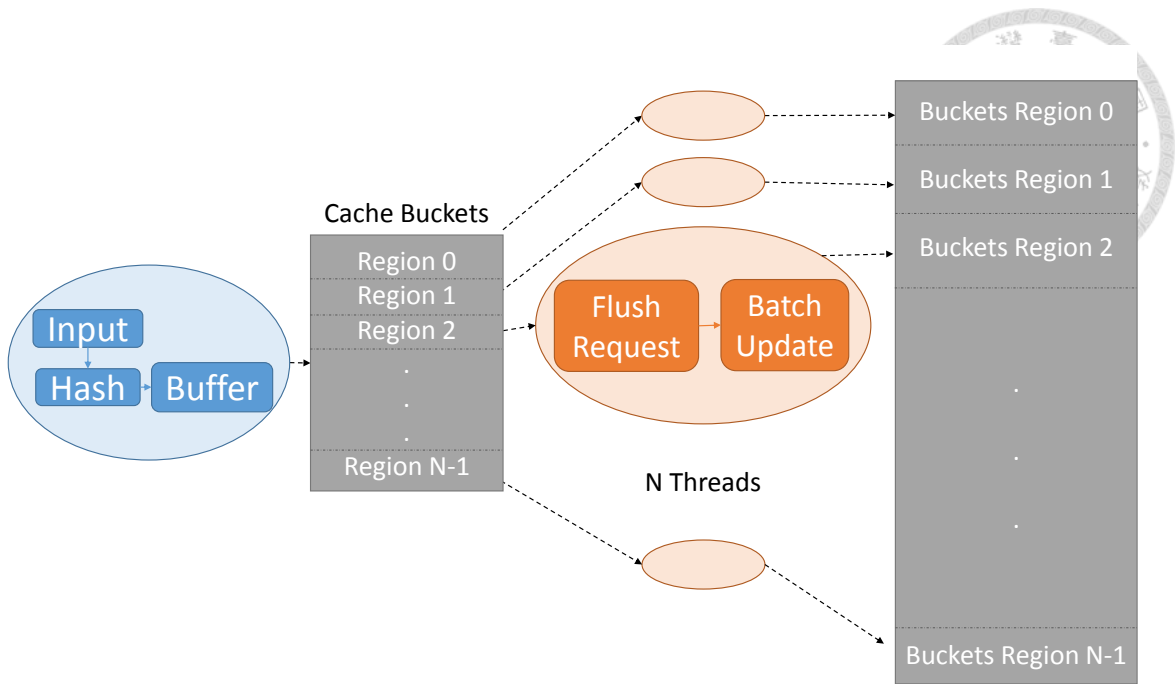


Figure 3.4: The structure of decoupled foreground-background cache-aware model.

regions, one for each background thread. To avoid costly lock contention for buckets, each background thread has disjoint responsible region of cache-buckets.

3.3.2 Flushing request queues

Since the cost of scanning meta-data about cache-buckets is too high, we use passive flushing threads. A flusher thread does not seek for cache-buckets who need to be flushed. It instead keeps checking a lock-free FIFO queue [11] contains "flushing requests" which are raised by the foreground thread, who can know if a cache-bucket requires flushing immediately. By checking the flushing request queue, the background threads can know which cache-bucket needs flushing, without additional operations.

But there is a load-balancing problem in this decoupled threading model. If the throughput of foreground and background threads cannot meet, there would be waste of computation powers, depends on the system scheduler. In our model, there are no other running threads who compete CPU cores with the foreground and background threads, and each thread has a dedicated logical core to avoid expensive thread switch. This makes fine-tuning the foreground and background threads throughput to obtain load balance for different workloads extremely hard.



Chapter 4

Experiments & Discussions

In this chapter, we are going to discuss our experiments, including our experiment environment, the experiment settings and discussions on experiment results.

4.1 Experiment Environment

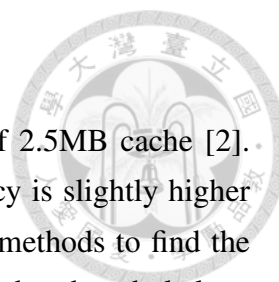
All experiments are compiled by `g++` with `-O2` flag and executed on the experiment platform with specifications listed in Table 4.1.

4.1.1 NUMA effect

The platform has two CPU sockets, with non-uniform memory access (NUMA) [10] in effect. We do not discuss NUMA in this work and would like to perform all experiments with minimum NUMA effects. However, if the program accesses the same data from different sockets, there must be cache-misses, since the last-level-cache of each socket is not shared across the sockets. So we have to bind the threads to cores on the same socket, and execute the experiment programs with `numactl`, forcing the loader to start the executable on the same node.

CPU	Intel Xeon E5-2620v2 [1]
# of Cores	6C12T
L3 Cache Size	15MB Shared
RAM	48GB
OS	Linux 2.6.32

Table 4.1: The specifications of experiment platform



4.1.2 NUCA effect

The last-level-cache of our experiment platform consists 6 slices of 2.5MB cache [2]. Each of the slices is shared by all cores on the socket, but the latency is slightly higher for a core accessing a far slice. There are researches [19] showing methods to find the address mapping with this organization. But in this work, we assume that the whole last-level-cache is uniformly available for the system's memory usage. Since the difference of latency between hitting local or remote slice does not exceed 5 cycles, we do not discuss effects of non-uniform cache access (NUCA) [8] in the experiments.

4.2 Experiment Settings

4.2.1 Data & request model

In our experiments, we assume all the incoming requests are fix-sized key-value pairs, and the size is the same as that of records in memory buckets. As Equation 3.3 shows, with the same available LLC capacity, the request size will determine nc , which represents how many requests the cache-buckets can hold in total. Furthermore, we assume there are only *UPDATE* requests. Before a test run starts, we fill up the memory buckets with a range of keys. During a test run, the system searches along a memory bucket for the corresponding record of an input request.

To minimize side effects and factors which we are not interested in for this work, we do not include *INSERT* workloads to avoid using extensible data structures which make dynamic memory allocation an important factor, while we do not focus on it. As for read workloads, we consider that the system can handle read requests with other CPU sockets. If strict read consistency is not required, the threads on other CPU sockets can read the memory buckets without interfering data cached by the CPU socket who handles update requests. However, this threading model is rarely implementable in our simplified input model, so we leave the mixed workload to future works.

4.2.2 Searching data structure

In order to find the position of corresponding record of an input request in a memory bucket, the system needs a search method for searching records in memory buckets, so the data table must be organized as some search structure. To keep our analysis model described in Chapter 3.1 not too complicated, we assume a memory bucket a memory block, and within each block the system uses linear search for experiments, with reasonable memory bucket size.

# of Threads	{1, 5, 10}
r	{8B, 16B, 32B}
a	{128B, 256B, 512B}
m	{200k, 400k, 600k, 800k}



Table 4.2: The general settings of various factors. There are total 108 cases.

4.2.3 General factors settings

For following experiments, if it is not specified, the factor settings of each experiment are those listed in Table 4.2. We take the average of results from 3 runs for each case.

4.3 Modeling the Cost per Input Request

For clear analysis of the factors in performance improvement of cache-aware model, we decompose the cost for processing an input request into two parts:

1. Base cost. This consists of all costs which are nothing to do with the memory buckets, such as accessing the input interface, calculating the corresponding bucket key of the record key. And for distributed baseline model, there is a lock cost. For cache-aware model, the base cost also includes costs of accessing the cache-buckets (one read & one write for each request) as its overhead.
2. Bucket cost. This is correlated to the configuration and status of the memory buckets. Since the RAM access is fine-grained and there is no asynchronous reading for RAM, the CPU cost of dealing with the memory buckets (e.g. key comparison) should be proportional to the memory cost accessing the memory buckets, with a constant cache-hit ratio. Moreover, according to Equation 3.4, with higher average per-request access size a , the memory cost increases faster in baseline model than in cache-aware model. In most cases, the cache-aware model has less cache-misses and thus has lower memory cost.

While having higher base cost due to additional cache-buckets accesses, the cache-aware model can offset its overhead, and even get lower total cost with advantage in bucket memory cost, if the bucket cost is high enough. We will discuss the actual values of such costs later.

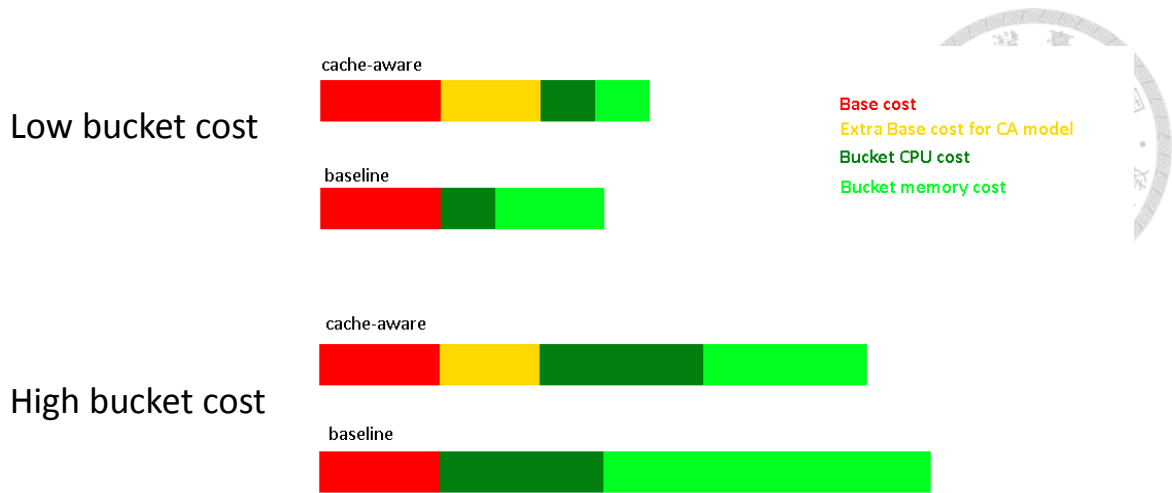


Figure 4.1: An illustration of request processing cost.

4.4 Workload Formation

4.4.1 Uniformly random

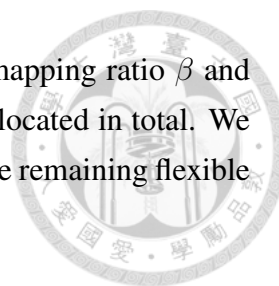
With uniformly random distributed requests, the per-request miss rate of cache-aware model and baseline is directly affected by the number of memory buckets. The more memory buckets, the higher miss rate for both models. However, if we want to keep the total size of memory buckets to avoid effects of different memory usage, we cannot apply m/nc factor analysis at fixed memory bucket length, which determines average per-request access size a , with uniformly random workloads.

However, in order to keep the memory layout constant for implementation, it is not desirable to change the actual allocated number of memory buckets. Thus we would like to use dynamic memory buckets mapping described as follows for experiments.

4.4.2 Clustered random

Instead of uniformly random, we would like to apply clustered random distribution of incoming requests. By activating only parts of all memory buckets are being accessed in a short period, we can get more flexible control on effective # of memory buckets, without changing the total size or unit size of memory buckets. The formation of clustered random is described as follow:

In a set of workload, the incoming requests are divided into several periods with identical length p , in unit number of requests. During each period, a map sized β determines the "active" memory buckets. For each cache-bucket, there are β memory buckets mapped to it, according to the map. The randomly generated requests during a period are with keys only belong to the mapped memory buckets.



Thus the effective memory bucket number m is determined by mapping ratio β and cache-bucket number n , no matter how many memory buckets are allocated in total. We can have reasonably many memory buckets and fixed bucket size while remaining flexible control on m .

4.5 Cache Partition using Coloring

As described above, the cache-buckets are expected to stay in LLC most of the time. However, in most CPU architectures, we have little explicit control of cache usage. If we let the memory allocation up to *malloc* system, it is possible that the not-frequently-accessed memory buckets compete cache capacity with frequently-accessed cache-buckets. To avoid such pollution behavior, we apply the concept of *pollute buffer* [21] for memory allocation. By partitioning the cache-buckets and memory buckets to different cache region, the memory buckets will compete only their cache partition and won't touch the cache-buckets region. Thus pollution to cache-buckets can be eliminated.

4.5.1 Side effect of CControl

For implementation, we use CControl[16] to partition cache-buckets, memory buckets and other metadata. With appropriate partitioning, the total cache misses can be reduced by 11% in average among all 108 cases listed in Table 4.2, in comparison to normal *malloc*. However, there are hidden side effects that cause lower throughput of colored version, even with less cache misses and page faults. So in later experiments we rather use the non-color version, since it has more consistent relative throughput while not having notably much more cache misses than colored version.

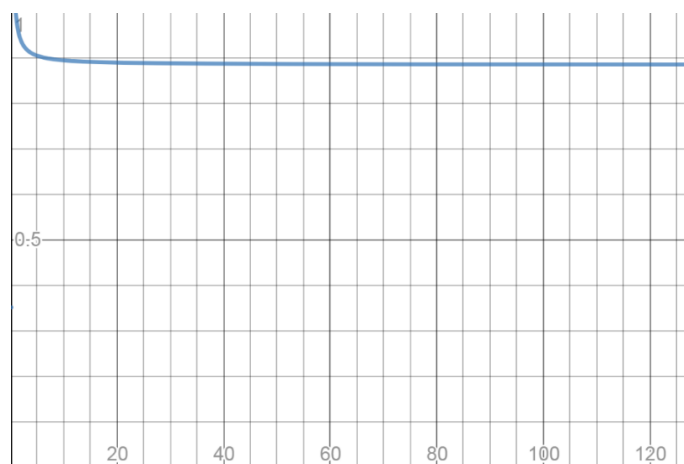


Figure 4.2: Plot Equation 3.2, M_c versus c . Let $m/nc = 4$.

Throughput	Few*Large	Medium	Many*Small
1 thread	5.87	5.42	5.17
5 threads	18.50	14.90	17.73
10 threads	34.27	28.48	32.80



Table 4.3: The average throughput of three cache-bucket organizations. $r=16B$, $m=400k$, $a=512B$.

4.6 Cache-Buckets Layout

In Equation 3.3, we let the total size of cache-buckets be fixed. Moreover, with the same request size r , the total number of cache-buckets capacity nc is constant. Thus we can have a few large cache-buckets, medium number of medium-sized cache-buckets or many small cache-buckets with the same nc . As Figure 4.2 shows, with fixed total capacity nc , the change in cache-buckets number and size has little effect on per-request miss ratio. So the effects of changing cache-buckets organization are limited to:

1. Meta-data size. In the implementation of cache-aware model, each cache-bucket has one lock and one counter. If the number of cache-buckets increases, the total size required for cache-bucket locks and cache-bucket counters will also increase.
2. On the other hand, if the number of cache-buckets decreases, the lock contention for multi-thread configuration could get heavier, since the probability of lock conflict is higher.

Since the two effects above are opposite, we operate experiments with 3 different cache-buckets organizations to find which effect matters more: For *few*large* organization, there are 12500 cache-buckets of 1kiB size each. For *medium* organization, we have 25000 cache-buckets, each is sized 512B. For *many*small* organization, the cache-buckets are allocated as 50000*256B. The results shows that *few*large* configuration win 105 out of 108 cases listed in Table 4.2. Table 4.3 and 4.4 shows the average throughput and average LLC-misses of the 3 cache-bucket organizations in some cases, respectively. Since the *few*large* organization has the best performance overall, we use this configuration as the default of cache-aware model for remaining experiments.

4.7 Overhead of Cache-Aware Model

In Chapter 4.3, we say that there could be higher base cost with cache-aware model. The extra accesses to cache-buckets for each request consists of cost of putting the request



Throughput	Few*Large	Medium	Many*Small
1 thread	7.98E+07	8.33E+07	8.76E+07
5 threads	7.58E+07	7.90E+07	8.17E+07
10 threads	7.42E+07	7.74E+07	8.04E+07

Table 4.4: The average LLC-misses of three cache-bucket organizations. $r=16B$, $m=400k$ $a=512B$.

into its corresponding cache-bucket and cost of reading it from the full cache-bucket upon flushing.

To measure the actual overhead of cache-aware model, we run experiments with no actual update to memory buckets actions included. For baseline model, the threads keep generating input requests. They then take and release the corresponding bucket lock without accessing memory buckets. For cache-aware model, the threads keep generating input requests, and then putting the record into the corresponding cache-bucket with locks. When a cache-bucket is found full, the thread who find it full directly set it to empty, without flushing the requests into memory buckets.

We run tests with 3 different record sizes. As shown in Figure 4.3, the base cost for handling a request of cache-aware model is slightly higher than that of baseline by about 70 cycles, no matter how the size of each record is.

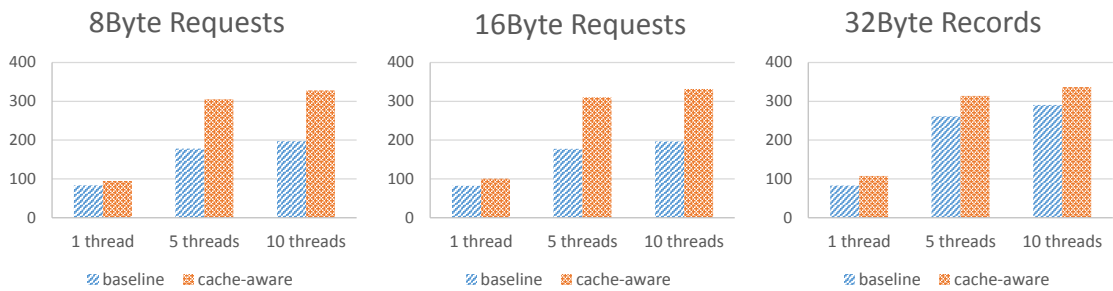


Figure 4.3: The average base cost for handling a request per worker thread, in terms of CPU cycles. The value is calculated by dividing the time elapsed by number of request handled per thread, and transforming the unit by multiplying the master CPU frequency.

4.8 Changing memory Bucket Size

With the extra costs, the cache-aware model won't win with low memory buckets cost per request, where the cache-aware model wins. If the size of each memory bucket is higher, the average required size of memory buckets per request a becomes larger, in order to seek for the existing record of the same key.

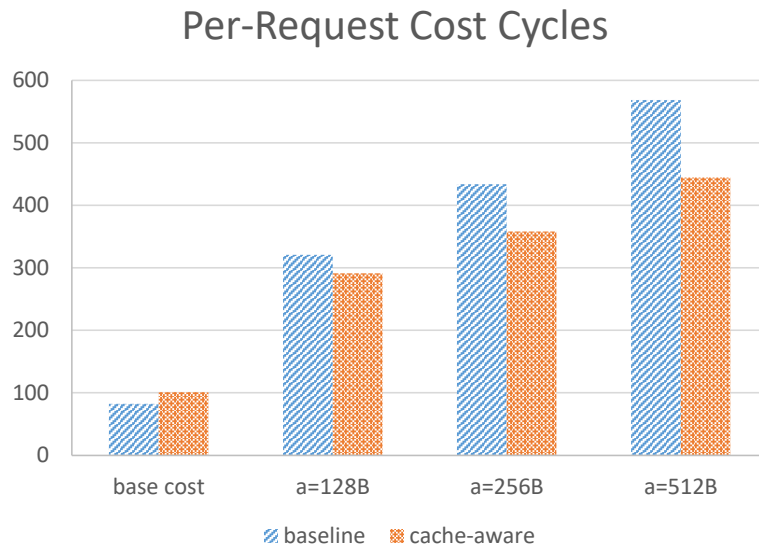


Figure 4.4: The average cost of handling a request, in unit of cycles. The effective # of memory buckets is 400,000 and the record size is set as 16Bytes.

But according to the simple random access test above, we know that long enough linear read can benefit a lot from hardware prefetching, and thus the advantage of cache-aware model over baseline in terms of cache-efficiency becomes smaller. So there should be an "advantage zone" of memory buckets sizes, which are large enough for cache-aware model to offset its overhead, and small enough that hardware prefetching helps not much of baseline model.

From Figure 4.4 we find that while having slightly higher base cost, the cache-aware model turns out to outperform the baseline since it has less cost increment as bucket size becomes larger. This suggests that cache-aware model performs relatively better when each request needs more memory access to deal with.

4.9 Changing Effective Memory Bucket Number

With the analytical prediction about miss-rate ratio, we expect that the relative last-level-cache misses of cache-aware model over baseline model will be high for few effective number of memory buckets m and will drop sharply to a knee point and then grow slowly with higher m . Since with higher m , the relative LLC-misses of cache-aware model is lower enough, we also expect that the cache-aware model would achieve higher throughput over baseline model. Figure 4.5 and 4.6 show the ratio of overall throughput and last-level-cache between cache-aware and baseline model, with 1 thread and $r = 16B$.

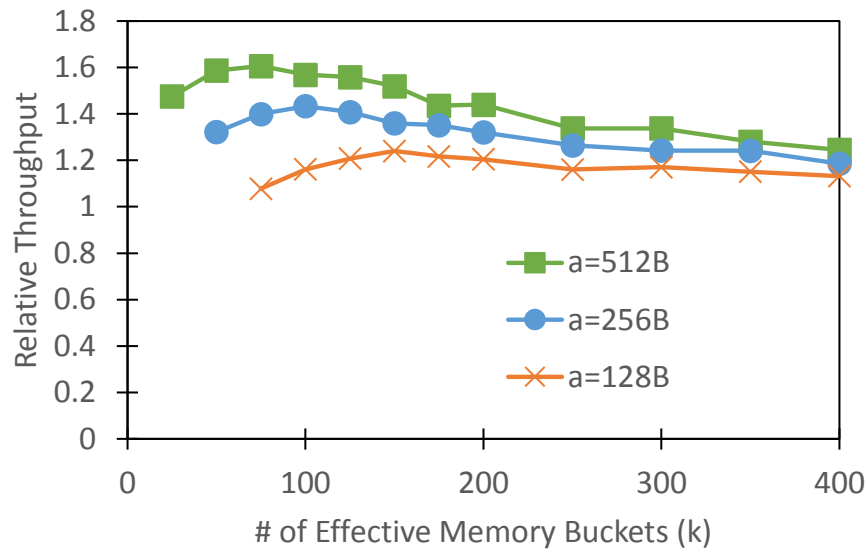


Figure 4.5: The relative throughput of cache-aware model over baseline, versus # of effective memory buckets.

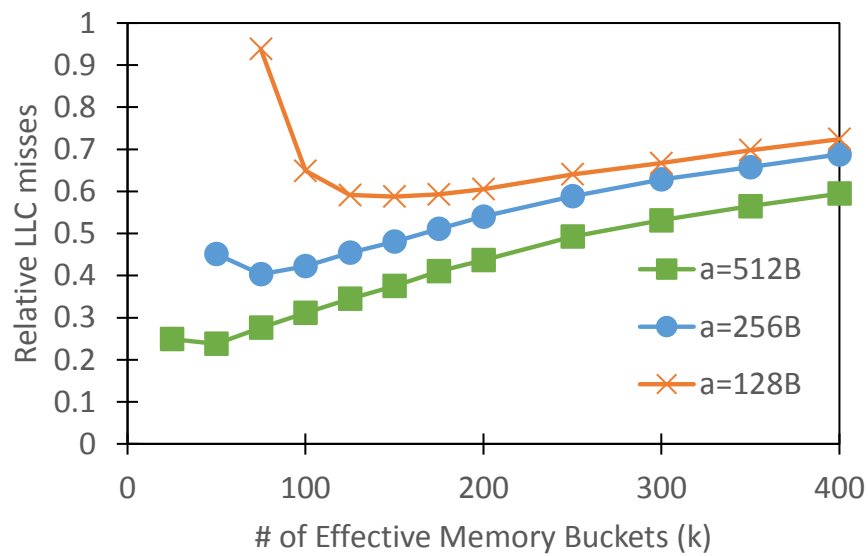


Figure 4.6: The relative LLC miss count of cache-aware model over baseline, versus # of effective memory buckets.

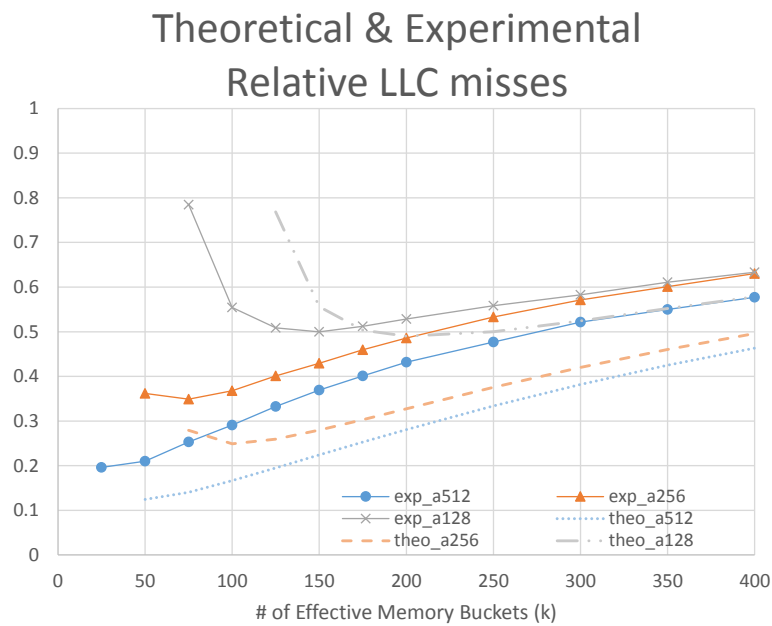


Figure 4.7: The experiment results of relative LLC misses of cache-aware model, compared with theoretical values in Equation 3.4.

4.10 Relative LLC Miss Count

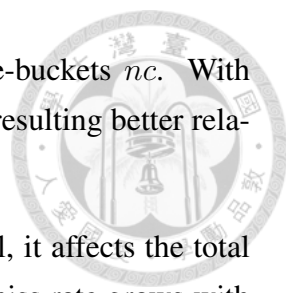
In modern CPU, the replacement policy is different from pure LRU. For example, recent Intel CPUs are equipped with generally improved policies such as RRIP [6]. Thus if the access pattern of buckets during updating the table isn't uniformly random (that is, somewhat predictable), the reuse rate can be higher than (2.1) with recent CPUs. But for uniformly random distribution, no extra control can help improve the hit rate.

On the other hand, most memory control units are equipped with hardware prefetchers, who can detect sequential access patterns and issue reads from the lower memory hierarchy before the next cacheline is required, avoiding cache-misses.

These factors are sources of error of theoretical relative LLC-misses shown in Figure 4.7. In addition, the model assume the cache-buckets occupy the whole last-level-cache, which is not true for implementation. However, while there are these implementation-dependent factors which are difficult to model, the theoretical prediction, especially the growing trend roughly meets experiment values and can be used for reference.

4.11 Summarization of All Factors

We would like to summarize all the factors discussed above. Figure 4.8 and Figure 4.9 show the relative throughput and LLC-misses of cache-aware model with 24 out of 108 combination of factors listed in Table 4.2. The factors are described as following:



1. The request size r . It determines the total capacity of cache-buckets nc . With smaller requests, the total capacity of cache-buckets is higher, resulting better relative cache misses and throughput for cache-aware model.
2. The average memory bucket access size a . For baseline model, it affects the total working set size. According to Equation 2.2, the per-request miss rate grows with a . On the other hand, since the miss rate of cache-aware model depends only on batch aggregation ratio in Equation 3.1, the cache-aware model can win further with higher a .
3. The number of threads. As shown in Figure 4.3, cache-aware model suffers more from thread interference because that the frequently accessed cache-buckets are shared across threads. Furthermore, since the total number of LLC-misses is constant to number of threads, the advantage in LLC-misses of cache-aware model will be split out by multiple threads. So the cache-aware model wins less in throughput with multiple threads.
4. The effective number of memory buckets m . This factor is discussed in Chapter 4.9.

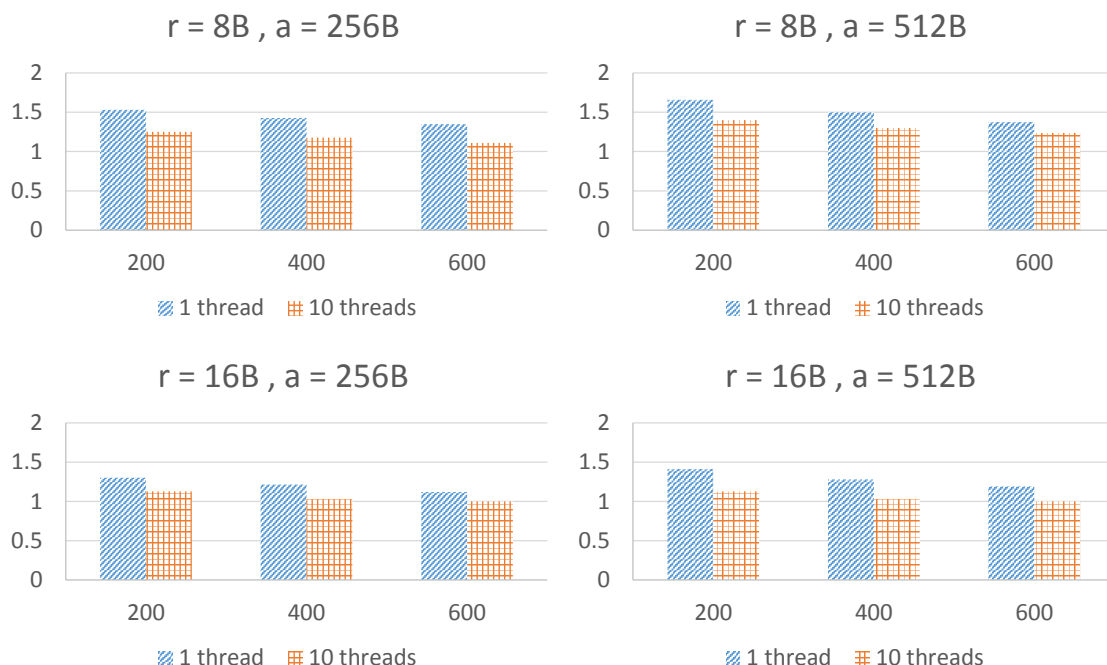


Figure 4.8: The bar charts show the relative throughput values of cache-aware model over baseline. The X-axis is effective # of memory buckets, multiplying 1000.

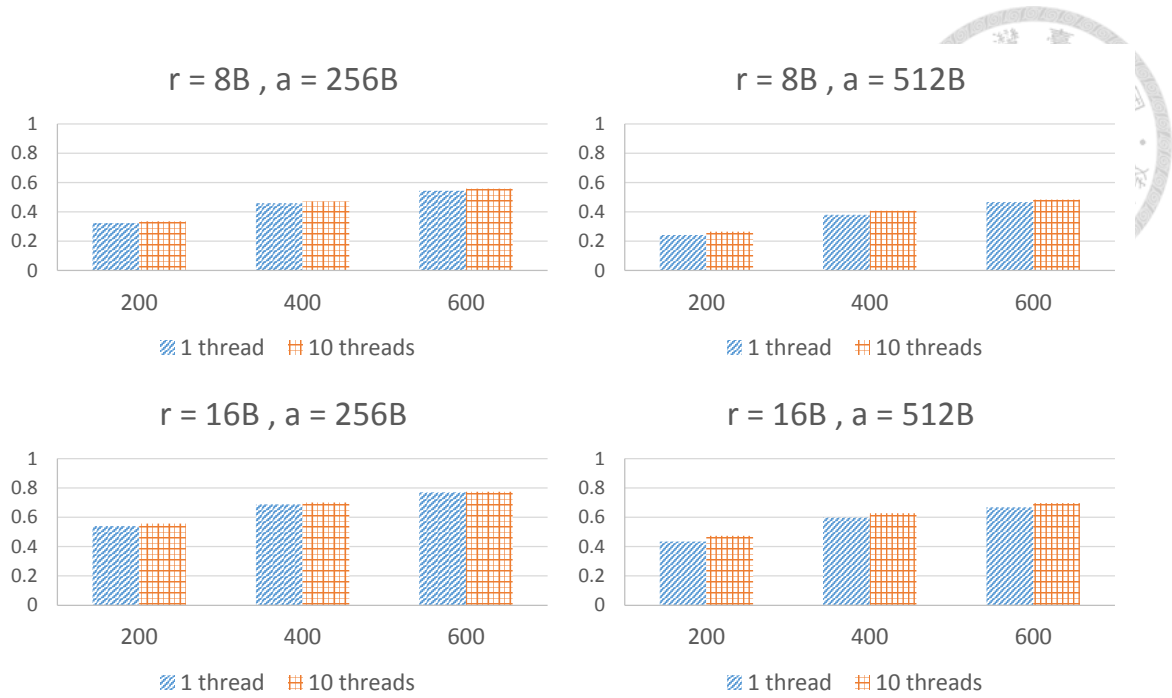


Figure 4.9: The bar charts show the relative measured LLC-miss counts of cache-aware model over baseline. The X-axis is effective # of memory buckets, multiplying 1000.

4.12 Improvement in Cost vs. Improvement in LLC-Misses

To understand the relationship between improvement in LLC-misses and improvement in overall performance, we apply linear regression on per-request cost cycle versus per-request LLC-misses for several sets of experimental results and formulate the cost caused by LLC-misses. Table 4.5 shows some sets of linear models. For $r = 16B$, the per-request LLC-misses of baseline model range from 4 to 11, which result in 57%-78% of total costs with 1 thread. Among these cases, the cache-aware model can save 20%-50% LLC-misses. However, since the cache-aware model has higher base cost and more expensive LLC-misses, the overall saved cost is less than 30% in all cases.

$r = 16B$	Cache-aware	Baseline
1 thread	$44x + 118$	$37x + 110$
5 threads	$68x + 209$	$60x + 136$
10 threads	$80x + 211$	$74x + 127$

Table 4.5: The fitted linear model of per-request cost cycles versus per-request LLC-misses. Each entry represents a model from data of 12 distinct (m, a) combinations.



Chapter 5

Conclusion

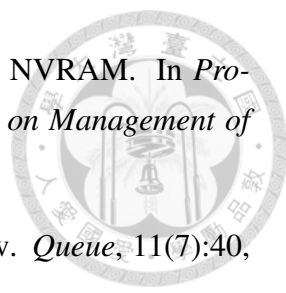
In this thesis, we propose a cache-centric update request handling model using a request buffering data structure called *cache-buckets*. This model targets to alleviate the poor cache utilization problem of in-memory DBMSs for low-locality update-intensive workloads. Our cache-aware batch update model tends to aggregate multiple update requests into one batched update to obtain higher temporal locality of cache usage, avoiding re-reference of memory data buckets. The experiment results show that the cache-aware model has up to 4 times less cache misses and 65% increase in throughput. Due to not negligible overhead, not dominating cache-miss penalties and limited room of improvement in LLC-misses, the cache-aware model can achieve only slight improvement of overall throughput.

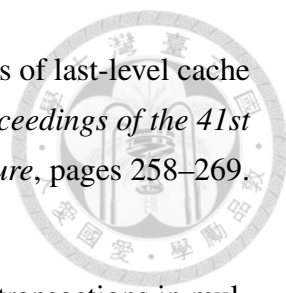
For future work, we would like to study the feasibility of designing a cache-centric, fault-tolerance storage system for NVRAM, which is equipped with the cache-aware batch update model, to achieve cache-speed request handling.



Bibliography

- [1] Intel® Xeon® Processor E5-2620 v2 (15M Cache, 2.10 GHz) Product Specifications. https://ark.intel.com/products/75789/Intel-Xeon-Processor-E5-2620-v2-15M-Cache-2_10-GHz.
- [2] Intel® 64 and IA-32 Architectures Optimization Reference Manual. <https://software.intel.com/sites/default/files/managed/9e/bc/64-ia-32-architectures-optimization-manual.pdf>, Jun 2017.
- [3] S. Chen and Q. Jin. Persistent b+-trees in non-volatile main memory. *Proceedings of the VLDB Endowment*, 8(7):786–797, 2015.
- [4] C. Diaconu, C. Freedman, E. Ismert, P.-A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwillig. Hekaton: SQL server’s memory-optimized OLTP engine. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 1243–1254. ACM, 2013.
- [5] J. Huang, K. Schwan, and M. K. Qureshi. NVRAM-aware logging in transaction systems. *Proceedings of the VLDB Endowment*, 8(4):389–400, 2014.
- [6] A. Jaleel, K. B. Theobald, S. C. Steely Jr, and J. Emer. High performance cache replacement using re-reference interval prediction (RRIP). In *ACM SIGARCH Computer Architecture News*, volume 38, pages 60–71. ACM, 2010.
- [7] T. Karnagel, R. Dementiev, R. Rajwar, K. Lai, T. Legler, B. Schlegel, and W. Lehner. Improving in-memory database index performance with Intel® Transactional Synchronization Extensions. In *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, pages 476–487. IEEE, 2014.
- [8] C. Kim, D. Burger, and S. W. Keckler. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. In *Acm Sigplan Notices*, volume 37, pages 211–222. ACM, 2002.

- 
- [9] H. Kimura. FOEDUS: OLTP engine for a thousand cores and NVRAM. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 691–706. ACM, 2015.
- [10] C. Lameter. Numa (non-uniform memory access): An overview. *Queue*, 11(7):40, 2013.
- [11] L. Lamport. Specifying concurrent program modules. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 5(2):190–222, 1983.
- [12] V. Leis, A. Kemper, and T. Neumann. The adaptive radix tree: ARTful indexing for main-memory databases. In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*, pages 38–49. IEEE, 2013.
- [13] J. J. Levandoski, D. B. Lomet, and S. Sengupta. The Bw-Tree: A B-tree for new hardware platforms. In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*, pages 302–313. IEEE, 2013.
- [14] Z. Majo and T. R. Gross. Memory management in NUMA multicore systems: trapped between cache contention and interconnect overhead. In *ACM SIGPLAN Notices*, volume 46, pages 11–20. ACM, 2011.
- [15] S. Pelley, T. F. Wenisch, B. T. Gold, and B. Bridge. Storage management in the NVRAM era. *Proceedings of the VLDB Endowment*, 7(2):121–132, 2013.
- [16] S. Perarnau, M. Tchiboukdjian, and G. Huard. Controlling Cache Utilization of HPC Applications. In *International Conference on Supercomputing (ICS)*, 2011.
- [17] H. Pirk, F. Funke, M. Grund, T. Neumann, U. Leser, S. Manegold, A. Kemper, and M. Kersten. CPU and cache efficient management of memory-resident databases. In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*, pages 14–25. IEEE, 2013.
- [18] H. Plattner and A. Zeier. *In-memory data management: technology and applications*. Springer Science & Business Media, 2012.
- [19] A. Scolari, D. B. Bartolini, and M. D. Santambrogio. A Software Cache Partitioning System for Hash-Based Caches. *ACM Transactions on Architecture and Code Optimization (TACO)*, 13(4):57, 2016.
- [20] D. N. Simha, M. Lu, and T.-c. Chiueh. An update-aware storage system for low-locality update-intensive workloads. In *ACM SIGPLAN Notices*, volume 47, pages 375–386. ACM, 2012.

- 
- [21] L. Soares, D. Tam, and M. Stumm. Reducing the harmful effects of last-level cache polluters with an OS-level, software-only pollute buffer. In *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*, pages 258–269. IEEE Computer Society, 2008.
- [22] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 18–32. ACM, 2013.
- [23] J. Yang, Q. Wei, C. Chen, C. Wang, K. L. Yong, and B. He. NV-Tree: Reducing Consistency Cost for NVM-based Single Level Systems. In *FAST*, volume 15, pages 167–181, 2015.
- [24] H. Zhang, G. Chen, B. C. Ooi, K.-L. Tan, and M. Zhang. In-memory big data management and processing: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 27(7):1920–1948, 2015.