FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

ASSIGNMENT OF MASTER'S THESIS

Title:	Efficient concurrent memoization system
Student:	Bc. Viacheslav Kroilov
Supervisor:	Ing. Daniel Langr, Ph.D.
Study Programme:	Informatics
Study Branch:	System Programming
Department:	Department of Theoretical Computer Science
Validity:	Until the end of summer semester 2020/21

Instructions

Get familiar with the problem of concurrent hash tables intended for efficient use in multi-threaded applications.
Get familiar with the problem of memoization systems, i.e., software cache databases with limited capacity and support for automatic data removal.
Design and implement an efficient concurrent memoization system with a focus on scalability for higher thread numbers.
Conduct an extensive experimental study of the proposed solution by its comparison with existing solutions.
Include the proposed solution into an existing HPC application and evaluate its efficiency. For implementation, use C++.

References

Will be provided by the supervisor.

Jan Janoušek
doc. Ing. Jan Janoušek, Ph.D.
Head of Department

[illegible]
doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague October 8, 2019

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master's thesis

Efficient Concurrent Memoization System

Bc. Viacheslav Kroilov

Department of theoretical computer science
Supervisor: Ing. Daniel Langr, Ph.D.

February 12, 2020

Acknowledgements

I would like to thank my parents, Mikhail and Elena, for the constant support and love they give to me.

I gratefully recognize the assistance of Daniel Langr, Ph.D. and doc. Ivan Šimeček with my academic research.

Also I would like to thank Dr. Adnan Aziz, Tsung-Hsien Lee, Daria Doronina, Elizaveta Ulyanova, and Anna Vasilenko for the invaluable feedback they gave on early versions of this thesis.

Declaration

Citation of this thesis

Kroilov, Viacheslav. *Efficient Concurrent Memoization System.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2020. Also available from: ⟨https://metopa.github.io/deferred_lru/⟩.

Abstrakt

Automaticky memoizační systém - také softwarová cache - ukládá omezený počet prvků v paměti, které byly nedávno zpřístupněny a zrychluje tak následný přístup k nim. Least-Recently-Used (LRU) je populární strategie nahrazování prvků pro hardwarovou a softwarovou cache. Nicméně, její paralelní implementace má nízkou škálovatelnost v důsledku přeskupování seznamu, které je prováděno jak při vyhledávání tak při vkládání.

V této práci je představena nová paralelní softwarová cache DeferredLRU, která vychází z LRU strategie. Hlavním cílem návrhu byla škálovatelnost a efektivní využití na mnoha-jádrových systémech. Toho bylo dosaženo použitím jiného řešení ke sledování pořadí přístupu k prvkům. Toto řešení podstatně snižuje počet opětovných vložení prvků do seznamu, což je hlavním faktorem zpomalení u běžné LRU cache.

Výkonnost a hit-rate DeferredLRU jsou citlivé na nastavení konfiguračních parametrů. Díky vyladělnému nastavení parametrů pro specifické vstupy byl dosažen vyšší hit-rate než u běžné LRU cache ve všech testovaných případech. Relativní rozdíl byl až 7,8%.

Výkon DererredLRU byl porovnán s existujícími alternativami, včetně souvisejícíh implementací cache z projektů Intel TBB a Facebook HHVM. Testované implementace cache byly hodnoceny až do 32 vláken (na 16 HW CPU jádrech). Při 32 vláknech, DeferredLRU bylo rychlejší ve všech 16 testech. Pokud byly přístupy distribuovány mezi více malých cache z důvodu lepšího paralelizmu (tzv. binning), DeferredLRU bylo rychlejší v 11 z 16 případů a ve zbylých 5 byl výkon blízko nejlepšímu pozorovanému výsledku. DeferredLRU s binning přístupem bylo až 28,8 krát rychlejší na 32 vláknech ve srovnání s jedno-vláknovým výkonem.

Klíčová slova LRU cache, memoizace, paralelní datové struktury, paralelní výpočet, softwarová cache, souběžný výpočet, strategie nahrazování prvků, techniky lock-free programování, víceprocesové systémy

Abstract

An automatic memoization system — also known as a software cache — stores a limited number of recently accessed elements and speeds up consequent accesses to them. Least-Recently-Used (LRU) is a popular replacement policy for hardware and software caches. However, its concurrent implementation suffers from high contention due to the list reordering performed both on lookup and insertion.

A novel LRU-inspired concurrent software cache, called DeferredLRU, is presented in this thesis. The main goal of the design was to make it scalable and suitable for many-core systems. These properties were achieved by using a different approach to tracking item access order. This approach substantially decreases the number of list reinsertions — the main factor of the contention in a regular LRU cache.

DeferredLRU throughput and hit-rate are sensitive to the meta-parameter setting. By fine-tuning meta parameters for specific inputs, it was possible to achieve higher hit-rate than of a regular LRU cache for every tested input. The relative difference was up to 7.8%.

DeferredLRU performance was compared to existing alternatives, including corresponding caches from Intel TBB and Facebook HHVM projects. Tested caches were evaluated with up to 32 threads (on a 16 HW cores CPU). In 32 threads evaluation, DeferredLRU was faster in all 16 tests. When accesses were distributed among multiple smaller caches for better parallelism (this approach is called binning), DeferredLRU was faster in 11 of 16 tests and was close to best-performing caches in 5 other tests. DeferredLRU with binning was up to 28.8 times faster on 32 threads compared to single-threaded performance.

Keywords Cache eviction strategy, concurrent computing, concurrent data structure, lock-free programming techniques, LRU cache, memoization, multi-processing systems, parallel computing, software cache

Contents

List of Figures

List of Tables

Introduction

Motivation

Modern computations and data processing rely heavily on caching systems. Software cache [1] (SW cache) is a key-value storage used to speed up access to commonly used or recently used elements in some domain.

Without caching, required values are recomputed or fetched from network each time they are requested. With a SW cache, the values are stored in memory once obtained. Later, the requested value is searched in memory before being retrieved again. If it is found, the costly recomputation of this value is omitted.

SW cache supports at least the following two operations: LOOKUP that looks up a value by its key in memory and INSERT that stores a new key-value pair (while possibly evicting some existing element). Usually, they are combined into one procedure that searches for the requested item, and in case it is not found the item is recomputed and inserted into the SW cache.

Among numerous SW cache applications, there are systems for optimizing server-side IO throughput [2, 3] and distributed file systems [4, 5]. Caching is an integral part of the database systems in many ways: it is used for managing disk buffering [6], server-side request caching, client-side ORMDB caches [7], and many more. Memcached [8] and Redis [9] are popular general-purpose caching solutions for modern network systems [10, 11].

An automatic memoization approach is a form of caching that is primarily used in algorithmization. This process is also denoted as tabling [12]. It was first described in [13], and it is widely used in term rewriting [14], artificial intelligence [15], and other scientific computations [16–18].

The SW cache is used when it is impractical or impossible to store all the values in local memory. Its goal then is to store the most valuable elements. Some stored elements must be evicted from the cache in order to free up space for the insertion of a new one. SW caches differ by the replacement strategy they use to choose such an element.

The *Least-Recently-Used* (LRU) [19] is a simple replacement strategy. When a replacement is required, the element that has not been accessed for the longest time is chosen. It is achieved by keeping all elements in a linked list ordered by the last access time. When an element in a cache is accessed, it is moved to the head of the list. The element at the tail of the list is considered least recently used, and it is chosen for eviction. The list is combined with a dictionary data structure (usually, a hash table) for fast item lookup by key.

This implementation performs well in a sequential program, but adapting this data structure for a concurrent environment raises multiple issues. One of the main flaws is the high contention on the list head since both LOOKUP and INSERT operations imply an insertion into the LRU list head. When each thread attempts to access the same memory address, these accesses are serialized. This limits the scalability of the whole cache.

What is more, it is hard to combine concurrent hash table and concurrent linked list in a single system while preserving the correctness and scalability of these data structures. There are known implementations for a doubly linked list and concurrent hash table (see Section 2.3.1). However, special care should be taken to avoid race conditions when combining them. For instance, an element can be evicted from the list while it is searched in the hash table. The routine that moves accessed elements to the list head must foresee this.

Contribution

An LRU-inspired concurrent SW cache, called DeferredLRU, is presented in this thesis. It attempts to preserve LRU caching efficiency while making it more suitable for a concurrent environment.

It uses a different approach to moving accessed elements to the head. Instead of being moved immediately after each element lookup, they are added to another linked list, called *Recent* list (in contrast to *LRU* list). When the number of elements in the *Recent* list hits some threshold, a single thread performs the PULLRECENT operation while other threads are still able to perform lookups and insertions concurrently. PULLRECENT evicts all the elements in the *Recent* list from the *LRU* list and reinserts them into the head. The whole sublist is inserted in a single step. This trick vastly decreases the total number of head insertions (making one insertion per PULLRECENT, not per lookup), and therefore, it improves scalability.

PULLRECENT is done as a part of cache consolidation. In case the cache is full, the consolidation stage also performs PURGEOLD operation. PURGEOLD evicts some tail nodes to reuse them for storing new items. It is guaranteed that *only one thread performs consolidation at a time*, and it is the only time when nodes are removed from the *LRU* list. The only other operation that is performed on the *LRU* list is the head insertion. These two facts allowed the list synchronization to be greatly simplified and reduced (see Section 2.3.2).

In addition to that, the thesis contains a comprehensive performance evaluation of DeferredLRU and compares it to existing concurrent LRU SW caches under different workloads.

DeferredLRU puts the main focus on improving cache scalability with a novel approach to updating the LRU list. The ideas presented in this thesis can be used for implementing other scalable concurrent SW caches that are based on more sophisticated replacement strategies (see Section 1.1) similarly to how LRU is used as a building block in many of these strategies.

Outline

The thesis proceeds as follows. In Chapter 1, an overview of selected SW caches and replacement strategies is presented. In Chapter 2, data structures and algorithms necessary for understanding the DeferredLRU implementation are introduced. In Chapter 3, several concurrent LRU SW caches are studied. These caches are later used in the performance evaluation for comparison with DeferredLRU cache. In Chapter 4, the DeferredLRU implementation is introduced. In Chapter 5, an extensive performance evaluation of DeferredLRU and other LRU containers is presented. It studies the effect of the DeferredLRU meta parameters choice and compares cache throughput and hit-rate under different workloads and with a varying number of threads. In Chapter 6, the future improvements to DeferredLRU are discussed.

Chapter 1

Related Work

In recent years, the progress on SW caches has been made in two primary directions: improving replacement strategy to make smarter evictions and improve performance, especially for parallel applications.

1.1 Replacement strategies

Least-Recently-Used (LRU) [19] is a simple replacement strategy. It always chooses to evict an element that has not been accessed for the longest time. Least-Frequently-Used (LFU) is another base approach to node eviction. It tracks the number of accesses to each element in the cache and evicts the one that has the least number of accesses.

Bélády's algorithm [20] is the most efficient replacement strategy. It discards entries that are not required for the longest time. However, to do so, the algorithm would have to see the input ahead, which is not possible with non-static inputs. Therefore, Bélády's algorithm cannot be implemented in reality. It is useful for evaluating other cache replacement strategies, as it sets the best possible hit-rate that can be achieved with a given input.

LRU-K [19] attempts to find a balance between both LRU and LFU approaches. It makes eviction decision based on last K accesses.

2Q [6] attempts to overcome an issue with inserted items that were not referenced afterward. Such items unnecessary take up valuable caches capacity. 2Q maintains two queues (hence the name 2Q). On first access, items are placed to the first FIFO queue. If an item is reaccessed before it is evicted, it is promoted to the second LRU queue. By changing the ratio between the queue capacities, it is possible to adjust eviction behavior.

Low Inter-reference Recency Set (LIRS) [21] uses the reuse distance metric — number of other elements accessed between two consecutive references to the given element. LIRS ranks entries in the cache based on the maximum of reuse distances between last and second-to-last reference and distance between the last access and current point in time.

Adaptive replacement cache (ARC) [22] attempts to exploit both recency and frequency of access locality. It maintains LRU and LFU caches and a set of ghost-entries (meta-data of recently evicted items) for each of them. The total cache capacity is dynamically divided between the two caches based on the number of hits each cache receives. That is, if a particular workload caches better with a frequency-based replacement strategy, the LFU cache capacity is gradually increased and vice versa. The adaptation process is continuous; therefore, ARC is able to capture changes in workload.

TinyLFU [23] is a cache admission policy. For a new element and a given eviction candidate, it decides whether it worths performing the replacement. It does so by considering access frequency that is tracked in a very economic way based on Bloom filter theory [24]. W-TinyLFU [23] is a cache replacement strategy that uses a combination of a small LRU cache and a much larger SLRU cache with an embedded TinyLFU admission policy. Initially, elements are written into the LRU cache and may be propagated to the SLRU cache on subsequent accesses.

Hawkeye [25] cache replacement algorithm is based on a completely different idea than the previous caches. It approaches the replacement decision as a binary classification problem. It executes Bélády's algorithm on a history of recent accesses to study the workload distribution and make predictions.

1.2 Concurrent caches

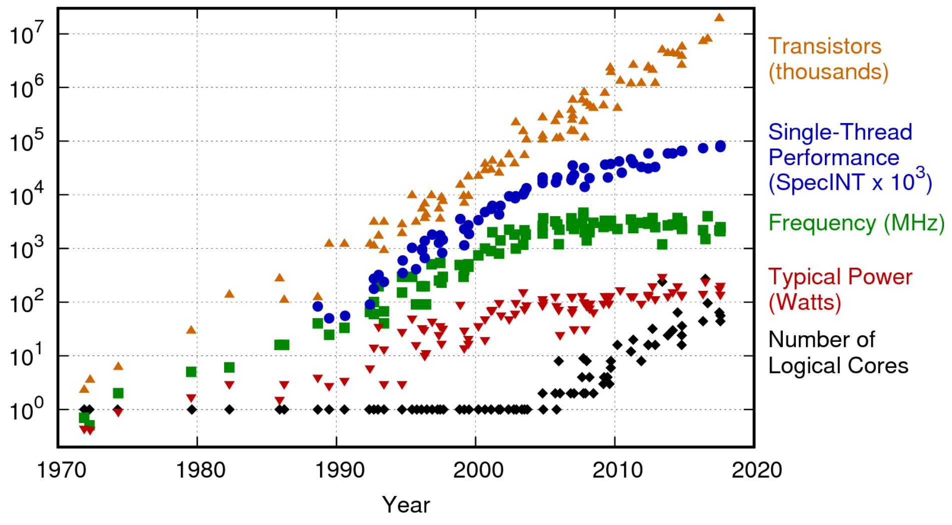
The demand for concurrent caching arises from the shift in CPU development trends. Gordon E. Moore has predicted that the number of transistors doubles every 2 years [27]. Figure 1.1 demonstrates that this statement remains correct for the last 5 decades.

However, single-threaded performance trend is not keeping up with this trend for the last 10 years. The overall performance is catching up by the number of logical cores, that has increased drastically even in consumer-grade processors. Server processors have been having tens of cores for a long time, and the counts approach hundreds rapidly.

There are several SW caches that are suitable for a concurrent program. C++ implementations of concurrent LRU cache can be found in the Intel Threading Building Blocks library [28] and the Facebook HHVM project [29]. These implementations are studied in Chapter 3.

Agnes [30] presents another concurrent cache, called BagLRU. Inserted items are accumulated in an "AgeBag" collection. When the AgeBag is full, another one is allocated, and new items are written to it. When the cache is full, the oldest AgeBag is emptied. Items that have been accessed since the corresponding was full are added to a newer AgeBag. The other items are evicted. Other concurrent cache implementations can be found in [31–33]. These implementations were written in Java. Therefore, they can not be directly compared with existing DeferredLRU implementation.

Figure 1.1: 42 years of microprocessor trend data [26]



Chapter **2**

Background

In this chapter, the algorithms, data structures, and concepts that DeferredLRU implementation is built upon are presented. At first, common data structures like linked list or hash table are introduced. It is followed by an overview of some approaches used in concurrent programming for ensuring thread synchronization. Then, a survey of existing concurrent hash tables is presented. Finally, a DeferredLRU-specific implementation of concurrent doubly linked list is introduced.

2.1 Sequential data structures and algorithms

2.1.1 Singly linked list

Singly linked list is a basic data structure for maintaining a sorted sequence of elements. It is built of nodes that consist of user data and a link to the next node. In most implementations, the last node link points to a distinctive value that is often denoted as NULL. List traversal stops when this value is reached. Another common option is to set the last link node pointing to the very first node. In this case, the list is called cyclic. The first linked list node is called the list head. Respectively, the last node is called the list tail. All other nodes can be reached from a head node. Usually, a linked list is handled by a link to its head. Sedgewick [34] provides a detailed explanation of the linked list data structure.

In comparison to other data structures, specifically arrays, linked list has the following advantages:

- Nodes in a linked list can be reordered by merely reassigning their next links. This *allows* $\mathcal{O}(1)$ *element insertion and removal in the beginning and middle* of a linked list in contrast to $\mathcal{O}(N)$ for array (when inserting into an array, all elements after the position of the insertion must be shifted and there is $N/2$ such elements on average).

- Consequently, this allows *fast list slicing and joining* operations. For instance, appending one list at the end of another is just a matter of setting the next link of the last node in one of the lists.
- Singly linked list has an *overhead of one link per element*, which is more than the overhead of an array, but still less than many other data structures.
- Linked list is a *stable container*, meaning that changing the order of list elements would not invalidate any existing link to a node inside the list.
- Linked list *can be intrusive*, meaning that it can be embedded into another structure. It is even possible to embed multiple linked lists in a single structure. Then instances of this structure may be linked independently through different lists while using the same memory placement.
- Linked lists can use a node pool for *efficient memory reclamation.* When a node is removed from a list, it is added to a dedicated linked list, called node pool, instead of deallocating the memory. The list is formed by reusing an existing next link in the nodes. Later, when required, the node is obtained from the node pool instead of allocating it in runtime.

  Adding and removing nodes from a node pool is expected to be much faster than using general memory management facilities. Some nodes may be preallocated and added to a memory pool beforehand to avoid initial allocations. Multiple lists of the same type may use a shared node pool.

However, these advantages come at a cost. One of the main downsides of a linked list is the absence of random access support. Accessing N-th node generally requires to iterate all the way down from the corresponding list head node. Moreover, traversing a linked list makes CPU cache futile. Since the next node can be anywhere in memory, each access may result in a CPU cache miss, pessimizing traversal performance. Finally, singly linked list does not allow backward traversal as nodes have no information about their predecessors. For the same reason, it is not possible to remove a node without knowing its predecessor. The last two problems can be solved with a doubly linked list.

2.1.2 Doubly linked list

Doubly linked list is a data structure that is in many ways similar to a singly linked list. However, instead of one link, each node in a doubly linked list contains links to both its preceding and following nodes (usually referred to as `prev` and `next` links). This modification allows doubly linked list to be traversed in both directions (but direct access by index is still not possible). Other than that, doubly linked list has the same advantages and disadvantages as singly linked list (see Section 2.1.1). See [34] for additional information.

2.1.3 Hash table

Hash table implements Dictionary abstract data type. It is capable of storing key-value pairs and looking up a value by the corresponding key. Also, it supports operations to add, update, and remove key-value pairs.

Internally hash table maintains an array of entries. It maps each key to an index in this array. To map a possibly infinite space of keys to a finite space of array indices, it uses a hash function [35]. A notable property of hash table is that its lookup time complexity does not depend on the number of items in it.

2.1.3.1 Hash collisions

In general case, the mapping cannot be injective, thus many keys are mapped to the same array index. This is called a *hash collision*. The number of collisions is controlled with a *load factor*. It is calculated as a maximum ratio of the number of items to the capacity.

There are two main approaches to dealing with collisions. With *closed addressing*, each entity in the hash table array is a linked list that contains all collided keys. With *open addressing* hash table array directly stores key-value pairs. A separate function for hash collision resolution is used. It generates a sequence of indices that are probed until an empty element in the array is found. If the sequence is $(N, N + 1, N + 2, ...)$, such probing is called linear probing. Sedgewick [34] explains these concepts in greater detail.

2.1.3.2 Cuckoo hash table

When using open addressing, it is unclear how may addresses are probed before the searched key is found. Cuckoo hashing is a variant of the open addressing hash table that guarantees the constant number of probes for lookup [36].

The insertion is performed as follows. For each key in the Cuckoo hash table, two positions in the array are calculated using different hash functions. These positions may be either in the same array or in two different arrays. During the insertion, both positions are checked. If at least one of them is empty, then it is chosen for insertion.

Otherwise, the collision is resolved by replacing one of the old elements with the new one and pushing the replaced elements to its second position. In case it is occupied as well, the element evicted in the previous step is written to the slot and the replacement is repeated for the newly evicted key.

If no empty slot was found after performing a predefined number of such replacements, the hash table is considered full, and it is rehashed.

Although insertion may result in a long chain of replacements, each key is always placed into one of its two positions. Therefore, the Cuckoo hash table allows a fast lookup, that has to check at most two memory locations.

2.1.3.3 Hopscotch hash table

The Cuckoo hash table requires a few memory accesses on lookup. These accesses can be anywhere in memory and each access likely results in a CPU cache miss. Hopscotch hash table [37] combines advantages of the Cuckoo hashing and linear probing to minimize CPU cache misses.

Items are kept in an array of buckets. H consequent buckets form a group of neighbors. Each bucket consists of a single slot for a key and the *hop information*. When a collision occurs, the key is attempted to be placed in one of the neighbor buckets. If the bucket is full, the hash table is rehashed.

The hop information is a bit array that has bits set for those positions in the group, that are occupied by the collided items. E.g., with $H = 6$, elements X, Y, and Z belong to the bucket 1. However, during collision resolution, they have been written to buckets 1, 2, 4. In this case, hop information equals to `110100`. Hop information is used to speed up lookups.

2.1.3.4 Cache-line hash table

Another attempt to improve memory access patterns concerning the CPU caching is made in CLHT [38]. It is based on a closed addressing. However, each bucket contains a chain of arrays of keys instead of separate nodes. Each array is of the same size as a CPU cache line. Therefore, each CPU cache transaction allows us to load and traverse multiple keys.

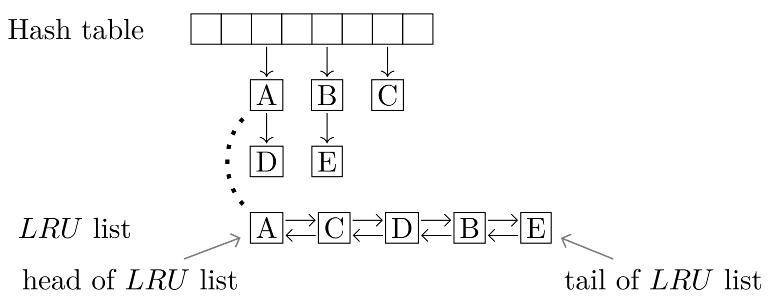
2.1.4 LRU cache

A typical LRU implementation is based on a combination of a hash table (for a fast element lookup) and a doubly linked list that keeps elements ordered by their access order, i.e., recently accessed and just added elements are inserted into the head and the least recently accessed elements are in the tail. Each successful element lookup results in the corresponding node being moved to the list head.

Figure 2.1 shows an example of LRU cache that contains 5 items (A, B, C, D, E). This implementation uses a closed addressing hash table, although other variants can be used as well. Item A has been accessed most recently. In case item B is accessed, *LRU* list will contain items in the following order: B, A, C, D, E. In case an eviction is performed, item E is removed from *LRU* list and hash table as it is the last entry in the *LRU* list.

The important property of this approach is that lookup and insert operations have $\mathcal{O}(1)$ space and time complexity. The lookup consists of the hash table lookup and optionally a linked list node removal and insertion. Insert operation consists of insertion into the hash table and into the linked list head. In the case of insertion into a full cache, another element also has to be evicted. This adds a hash table removal and a linked list removal. Indeed, all operations are $\mathcal{O}(1)$.

Figure 2.1: LRU cache structure



2.2 Approaches to synchronization in concurrent programs

This section describes some general approaches to thread synchronization, primitives that are used by these approaches, and common pitfalls that may occur in concurrent programs.

2.2.1 Race conditions

As described in Section 1.2, the main advance in modern chip development is made towards increased parallelism. Hence nowadays, many programs are required to support multi-core execution, and so do their building blocks, namely data structures and algorithms.

For most sequential codes that actively interact with memory, it stands that executing them in parallel results in undefined behavior and often a program crash too. When multiple threads access shared memory, they must appropriately synchronize reads and writes to the memory. Therefore, the corresponding codes must be rewritten with concurrency in mind. In other words, with the idea that other threads may execute any other viable code at the same time.

It is easy to demonstrate the issues coming up with multi-threaded execution even with a simple function that increments an integer X. On a machine instruction level, its implementation may look like in Algorithm 1. The important part is that the increment is performed in 3 steps: load current value from memory, increment it, write it back to memory.

In case two threads make progress on the function simultaneously, it is expected that X is incremented twice. However, due to the nature of concurrent programming, the instructions of different threads may be executed in any relative order. It may happen that on a global timeline, the instructions are ordered the way it is shown in Table 2.1. One can see that both threads concurrently read and increment the initial value of X. Therefore, after execution, X is only incremented by 1. Such a circumstance is called a *race condition.*

Algorithm 1 Integer increment

```
function INCREMENT($X$)
    Load $X$ to register
    register ← register + 1
    Write register to $X$
end function
```

Table 2.1: Integer increment race condition

Thread A	Thread B	X	$R0$	$R1$
		0	?	?
Load X to $R0$		0	0	?
	Load X to $R1$	0	0	0
	$R1 \leftarrow R1 + 1$	0	0	1
	Write $R1$ to X	1	0	1
$R0 \leftarrow R0 + 1$		1	1	1
Write $R0$ to X		**1**	1	1

The ordering presented in Table 2.1 may appear unlikely to happen. However, in a similar test, where two threads incremented a shared variable 100000 times, the final value of the variable is 120937 instead of 200000. About 40% of increments had been discarded due to the race condition in the code. In practice, it turns out that Murphy's law [39] applies to concurrent programming particularly well.

For eliminating the race condition discussed above, threads must be prevented from incrementing the same value and overwriting each other changes. Several approaches to such synchronization are presented in this section.

2.2.2 Mutual exclusion lock

Many concurrent data structures often rely on a mutual exclusion lock (mutex) for ensuring thread synchronization. Mutex prevents a part of the algorithm from being executed by multiple threads at the same time. Such a part is usually called a *critical section*.

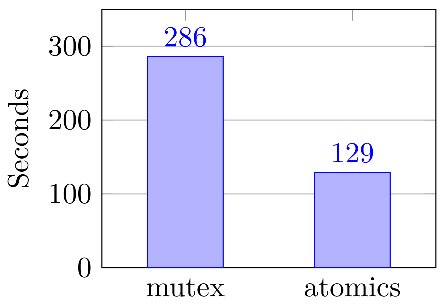
Mutex is supposed to be locked when entering the corresponding critical section and unlocked upon exit. When some thread attempts to lock an already locked mutex, it is suspended until the mutex is unlocked. Additionally, mutex provides non-blocking TryLock operation that attempts to lock the mutex, but the thread is not suspended if the mutex is locked already.

The 3-step number increment in Algorithm 1 is an example of a critical section. It may be implemented with a proper mutex-based synchronization as shown in Algorithm 2. In case of the ordering denoted in Table 2.1, thread B would be suspended until A unlocks the mutex.

Algorithm 2 Integer increment with mutex synchronization

```
function INCREMENT($X$, $mutex$)
    LOCK($mutex$)
    Load $X$ to $register$
    $register \leftarrow register + 1$
    Write $register$ to $X$
    UNLOCK($mutex$)
end function
```

Figure 2.2: Concurrent integer increment from 0 to 3200000000 with 32 threads using different synchronization approaches



2.2.3 Atomic operations

Many modern processors support atomic instructions [40,41]. These operations are different from regular ones because they are executed in a single memory transaction even though they consist of multiple memory loads and stores. This means that other threads will either see the memory before the operation starts or after it finishes, but never in some intermediate state.

One of the basic atomic operations is ATOMICCAS(A, C, $value$) (atomic Compare-And-Swap). If the current value of variable A equals to the value of variable C, A is set to $value$. Otherwise, the current value of A is written to C, but A itself remains unchanged. The operation returns a true/false result that tells whether $value$ was written to A. The atomicity ensures that the value of A cannot be changed concurrently after it was observed by a thread performing ATOMICCAS until this operation finishes. Full ATOMICCAS semantic is denoted in Algorithm 3.

Other common atomic operations are ATOMICEXCH and FETCHANDADD. ATOMICEXCH(A, $value$) operation returns the current value of A, while atomically setting A to $value$. FETCHANDADD(A, $value$) fetches the current value of A while incrementing A by $value$.

Many atomic operations can be expressed with ATOMICCAS. For example, ATOMICEXCH implementation based on Compare-And-Swap is presented in Algorithm 4.

Algorithm 3 Atomic Compare-And-Swap

```
function AtomicCAS($A$, $C$, value)
  Output: returns TRUE if value has been written to $A$, FALSE otherwise
    atomically
        if $A = C$ then
            $A \leftarrow value$
            return TRUE
        else
            $C \leftarrow A$
            return FALSE
        end if
    end atomically
end function
```

Algorithm 4 Atomic Exchange

```
function AtomicExch($A$, value)
  Output: returns the last value of $A$ before assignment
    $tmp \leftarrow A$
    repeat
    until AtomicCAS($A$, tmp, value)
    return tmp
end function
```

Atomic operations are often used instead of critical sections with mutexes because they may yield better performance. In a small test of incrementing an integer 3200000000 times with 32 threads version with atomic operations is 2.2x times faster than the one with a mutex (see Figure 2.2). For details on the test machine, see Chapter 5.

2.2.4 Coarse-grained synchronization

The simplest way to make any data structure thread-safe is to add a single mutex that guards all its operations. This approach is a so-called coarse-grained synchronization. The main advantage of it is its simplicity. Indeed, all algorithms are the same with slight modification for mutex locking.

However, only one thread may operate on a data structure at a time. Hence, the scalability is limited. If the data structure is the central component of some computations, then running these computations on multiple cores would show almost no speedup compared to a single-threaded run. The performance often even degrades with the number of threads rising.

2.2.5 Fine-grained synchronization

The scalability can be improved if the data structure operations are divided into multiple independent critical sections. Then different threads can execute the operations in parallel if they are in different critical sections. They still may happen to enter the same critical section at the same time, in which case one of them is suspended. However, this is not expected to happen too often. This approach is called fine-grained synchronization.

In the case of LRU cache, each operation requires the accessed element to be moved to the head of the *LRU* list. Therefore each thread enters the same critical section (associated with the *LRU* list head) on every Lookup and Insert operation. This limits the applicability of fine-grained synchronization to LRU cache.

2.2.6 Binning

For some data structures, scalability can be improved by *binning*. The trick is to allocate multiple smaller instances of a data structure — bins — and to distribute concurrent accesses among them. The number of bins is to be derived from the expected number of threads.

In application to SW cache, a binning adapter distributes keys uniformly across all bins using a hash function. Since most SW caches internally use a hash table, the binning adapter and the internal hash table must use different hash functions.

Binning significantly improves scalability even with coarse-grained synchronization (see Chapter 5). On the other side, it unpredictably affects cache hit-rate, improving it in some cases and worsening in others (see Appendix B). Also, it may have some side-effects like worse CPU caching (since multiple SW caches combined have a larger memory footprint).

2.2.7 Lock-free programming

Lock-based design is prone to system-wide stalls in case a thread that is holding a mutex is suspended (e.g., by the operating system scheduler). If a thread never releases the lock, other threads may stall indefinitely long. For instance, this may happen with priority inversion when the process scheduler keeps granting time to high-priority threads that are impeded by a mutex locked by a low-priority thread. Therefore, the system as a whole does not make progress at all.

In contrast to this, a lock-free programming model guarantees system-wide progress. That is, there is always at least one thread that can make progress even if all other threads are suspended. In practice, this usually means avoiding using mutexes and using atomic memory operations for thread synchronization instead. Fraser [42] provides a more in-depth explanation of lock-freedom and related concepts.

2.2.8 ABA problem

ABA problem is a type of a race condition, that mostly happens in atomics-based synchronization. It occurs when a thread relies on the assumption that two consecutive reads from a specified memory location being equivalent implies that the dependent data structure has not changed.

In reality, another thread may overwrite the checked variable multiple times, make some other changes to the dependent data structure, and by coincidence, write the initial value to the checked variable in the end. The first thread then would not detect the occurred changes since the checked variable is effectively the same. Consecutive operations performed by the thread results in undefined behavior and likely in the data structure corruption.

ABA problem can be easily demonstrated with a naive linked list-based lock-free stack implementation. Stack is a data structure with two basic operations: Push adds objects to a stack, and Pop extracts them in reverse (last in, first out) order.

The simple implementation presented in Algorithm 5 is prone to the ABA problem, as shown in Table 2.2. One can see that after the execution finishes *stackHead* points to invalid memory location.

Dechev et al. [43] introduce a technique for systematic ABA problem avoidance and compare it with other existing approaches. DeferredLRU avoids the ABA problem by putting specific constraints on the used lock-free data structures that make it impossible to happen.

For example, a constraints that could prevent ABA problem in Algorithm 5 is to avoid using Pop operation at all. The ABA problem cannot emerge from Push as this operation does not rely on the stack not being changed apart from the *stackHead* variable itself. By using some other operation for resetting the stack (see Section 4.1.6), the ABA problem can be avoided without using any explicit ABA avoidance technique.

Algorithm 5 Lock-free stack PUSH and POP (prone to ABA problem)

```
1: global $stackHead \leftarrow$ NULL
2: function PUSH($node$)
3:     repeat
4:         $currentHead \leftarrow stackHead$                        ▷ Atomic load
5:         $node.next \leftarrow currentHead$
6:     until ATOMICCAS($stackHead, currentHead, node$)
7: end function
8: function POP
9:     repeat
10:        $node \leftarrow stackHead$                               ▷ Atomic load
11:        if $node =$ NULL then
12:            return NULL                                          ▷ Stack is empty
13:        end if
14:        $nextNode \leftarrow node.next$
15:    until ATOMICCAS($stackHead, node, nextNode$)
16:    return $node$
17: end function
```

Table 2.2: ABA problem in a lock-free stack

	Thread A variables	
Stack	*node*	*nextNode*
Initial state $stackHead \rightarrow X \rightarrow Y \rightarrow Z \rightarrow$ NULL	?	?
Thread A executes Pop until being suspended between lines 14 and 15 $stackHead \rightarrow X \rightarrow Y \rightarrow Z \rightarrow$ NULL	X	Y
Thread B executes Pop; reference to X becomes invalid $stackHead \rightarrow Y \rightarrow Z \rightarrow$ NULL	X	Y
Thread B executes Pop; reference to Y becomes invalid $stackHead \rightarrow Z \rightarrow$ NULL	X	Y
Thread B executes Push and inserts X' **New node X' happens to have the same address as X** $stackHead \rightarrow X' \rightarrow Z \rightarrow$ NULL	X'	Y
Thread A finishes Pop **AtomicCAS on line 18 succeeds as the pointer value is the same** $stackHead \rightarrow Y \rightarrow ?$	?	?

2.3 Concurrent data structures and algorithms

2.3.1 Concurrent hash table

Hash table is a key data structure in numerous applications. There is a substantial progress made towards fast concurrent implementations. Chen et al. [44] and Maier et al. [45] present the most recent studies on concurrent hash tables.

Specifically, Chen et al. [44] conclude that there is no silver bullet: a concrete hash table must be chosen depending both on the typical use case and the target hardware (as the implementations often rely on specific hardware properties).

2.3.1.1 Fine-grained synchronization

As with other data structure, any hash table can be made concurrent using a coarse-grained synchronization approach. However, due to the nature of hash tables, accesses are usually distributed uniformly across all used memory. It makes the hash table a good fit for a fine-grained synchronization.

Indeed, the fine-grained hash table implementation is straightforward. Each bucket is associated with its mutex. Each thread locks the corresponding mutex before accessing nodes. This way, different buckets can be accessed concurrently. Considering the fact that hash tables usually contain many buckets, the contention on each of them is expected to be low.

This design results in large memory overhead, since each mutex may take up a considerable amount of memory (for instance, on Linux platform each mutex takes up 40 bytes). It can be optimized by associating one mutex with a group of buckets. With many groups, there is still enough independent critical sections. At the same time, the memory overhead is divided across all buckets in a group.

Locking mutex is an expensive operation. Therefore, such design usually yields low per-thread performance. What is more, if many threads attempt to access the same value at the same time, all the accesses are serialized. Never the less, many concurrent implementations rely on this kind of lock-based design. Other implementations are based on the lock-free design.

2.3.1.2 Concurrent Cuckoo hash table

Scouarnec [46] presents a scalable concurrent version of the Cuckoo hash table, called Cuckoo++. It uses lock-based design with one lock per bucket. Cuckoo++ attempts to omit a costly fetching of the secondary location by using the following heuristics.

Firstly, it relies on an “optimistic approach” — it expects the element to be found in its primary location and therefore it avoids memory prefetching for the secondary one. Secondly, each primary location bucket has a Bloom filter [24], that is able to detect, that the key is definitely not in the secondary location without actually accessing it.

2.3.1.3 Concurrent Hopscotch hash table

Herlihy et al. [37] describe both sequential and concurrent versions of the Hopscotch hash table. Concurrent Hopscotch also employs an optimistic partially lock-free synchronization. Each bucket maintains its lock and a version number, that is associated with all the keys mapped to the bucket (even if they are placed into different buckets due to collisions). Insertions and evictions use the lock for mutual synchronization and updates the version counter after each change.

However, LOOKUP optimistically expects that the bucket is not changed while it is accessed. It records the version number in the beginning and compares it with a current value after the lookup. If the values are equal, then the bucket has not been changed concurrently and the data is valid. Otherwise, the operation is repeated on its slow path that considers concurrent modifications.

2.3.1.4 Concurrent Cache-line hash table

There are two concurrent versions of CLHT [47]. The first one, called CLHT-LB, uses lock-based synchronization. Each array of buckets contains an additional lock, that is held when accessing the data.

The second version — CLHT-LF — is a lock-free hash table. It relies on atomic version-data pairs for detecting concurrent changes. When the data is modified, the version is increased. Then the updated pair is attempted to be posted to the shared memory using ATOMICCAS.

2.3.1.5 Implementations

These are existing high-performance hash table implementations [28, 47–52].

DeferredLRU uses a simple closed-addressing fine-grained hash table (as described in Section 2.3.1.1). The reason behind this is that the main focus was put on the *LRU* list and *Recent* list synchronization. Using a simpler hash table helped to avoid many bugs. The possibility of using a more sophisticated concurrent hash table is a concern of future research.

2.3.2 Lock-free doubly linked list for DeferredLRU

Several DeferredLRU routines insert a node or a sublist into a doubly-linked list head (elements are inserted into *LRU* list head to denote that they have been accessed recently). In order to simplify such insertion, a dummy head node is introduced, and inserting into *LRU* list head means inserting right after this dummy head.

To make this operation efficient, a lock-free list insertion routine (see Algorithm 6) has been developed for DeferredLRU. Note that single element insertion is generalized to list insertion (since one list node is a list itself).

Algorithm 6 Lock-free list after-head insertion

```
procedure DListAtomicInsert($head, first, last$)
    $first.prev \leftarrow head$
    $oldNext \leftarrow head.next$
    repeat
        $last.next \leftarrow oldNext$
    until AtomicCAS($head.next, oldNext, first$)
    $oldNext.prev \leftarrow last$
end procedure
```

The insertion is performed in the following steps:

1. The previous link to the first element is set to the head element (as it follows the head after the insertion).

2. The next link to the head element is set to the first element, while its previous value is written to the next link of the last element. Since this operation cannot be performed atomically, it is done with a CAS loop (using AtomicCAS operation).

3. The previous link to the old next element is updated.

This routine is not valid for a general concurrent doubly linked list, because it may result in a corrupted list state if a concurrent removal is performed. However, it is safe to use if the following constraints are ensured:

1. All insertions are performed only right after the head element.

2. No other thread may change the sublist that is currently being inserted.

3. No thread can remove an element if its previous link points to the list head.

It is critical to update the previous link of the old next element as the very last step. This way the race condition when the old next element is concurrently removed during the insertion is avoided. A node pointing to the head acts as a virtual boundary. It divides the parts of the list that are affected by insertions and by removals. It is prohibited to remove this node (see constraint 3). As soon as the previous link is updated, all other links (namely, *last.next*) are already correctly set up, and no other thread would update these nodes (as insertions are performed right after the head, and only one thread may perform a removal at a time). Therefore, from now, it is safe to remove this node.

Chapter 3

Concurrent LRU Caches

There are several concurrent implementations of an SW cache based on the LRU replacement strategy. The following implementations are chosen for comparison with the DeferredLRU implementation.

3.1 LRU with mutex

A sequential LRU cache implementation can be easily adapted to a concurrent environment using a coarse-grained synchronization approach (as described in Section 2.2.4). A mutex is added to the cache structure. Each operation is amended with a mutex locking in the function prologue and unlocking after the operation.

With this setup, only one thread may access the cache at a time. On the other side, hash table and *LRU* list operations are performed in a single critical section. This greatly simplifies synchronization as there is no need to consider inconsistencies, when a node is presented in one data structure, but it is missing in the other one. It may be an acceptable trade-off between scalability and implementation complexity for some applications.

3.2 ConcurrentLRU

In application to LRU cache, the fine-grained synchronization can be applied as follows. Both cache components — the hash table and the linked list — are replaced with their concurrent implementations. Different hash table buckets can be accessed concurrently. Each hash table bucket has its mutex that is locked when the bucket is accessed. *LRU* list elements can be accessed in parallel as well. Since LRU operations like Lookup and Insert access both components, they should be carefully synchronized and accessed in a way that no deadlock occurs. Complete ConcurrentLRU implementation is available at [53].

Each LRU node has its independent lock that is used for synchronization inside *LRU* list. In addition each node contains presence binary flags: one for each hash table and *LRU* list. Using an actual mutex per node would be too expensive, so the lock is encoded together with the presence flags and a hash table pointer in a single 64 bit machine word using `folly::PackedSyncPtr` from folly library [48]. It is guaranteed that node content can not change while it is locked.

Insert consists of the following steps. At first, an empty node is obtained (either from a node pool or from cache eviction in case it is full) and initialized with new key-value pair. Then it is inserted into the *LRU* list. After this it is attempted to be inserted into the hash table. This may fail in case another node with the same key has been inserted already. In this case, the node is attempted to be removed from *LRU* list (it could have been evicted by another thread already), and then it is returned to the empty node pool.

Lookup consists of the following steps. The key is looked up in the hash table. In case it is found, the corresponding key is attempted to be shifted to the *LRU* list head. It may not be possible in case the node has been just evicted from the *LRU* list.

3.3 Intel TBB LRU

Intel Threading Building Blocks [28] is a concurrent framework from Intel. “The library provides a wide range of features for parallel programming, including generic parallel algorithms, concurrent containers, a scalable memory allocator, work-stealing task scheduler, and low-level synchronization primitives.” [28].

One of the containers that Intel TBB provides is a concurrent LRU cache. Its design is based on a different variant of a coarse-grained synchronization. Each thread records operations that it is willing to execute to a shared worklist (`aggregator` in the Intel TBB terminology), but there is only one thread at a time that executes these operations. Therefore, all operations on the data structure itself are inherently sequential, and only the worklist has to be synchronized. DeferredLRU uses a similar trick for maintaining *Recent* list and schedule cache consolidations (see Section 4.3.1).

The worklist is implemented as a lock-free LIFO stack that is based on a singly-linked list that supports concurrent head insertion and concurrent list slicing.

The cache itself is implemented with the data types from C++ Standard Template Library. It uses a non-intrusive `std::list` for *LRU* list and `std::map` for the item lookup.

3.4 Facebook HHVM LRU

HHVM [29] is a virtual machine for faster Hack and PHP code execution developed by Facebook. HHVM source code contains another concurrent LRU cache implementation. Its design is based on coarse-grained synchronization and binning (bins are named *shards* in the HHVM terminology).

Each shard is represented by a combination of a concurrent hash table (HHVM LRU uses `tbb::concurrent_hash_map` from Intel TBB [28]) and a doubly linked list with a mutex-based coarse-grained synchronization.

LOOKUP and INSERT algorithms used in HHVM are presented in Algorithm 7 and Algorithm 8. One particular trick that vastly improves scalability is the fact that an accessed element is not always moved to the *LRU* list head — if another thread has locked the *LRU* list mutex, the thread skips the move entirely. This may result in sub-optimal cache hit-rate, but this allows LOOKUP to be non-blocking.

Algorithm 7 Find element in HHVM cache

```
function LOOKUP(key)
    hashNode, success ← FINDKEYINMAP(key)
    if not success then
        return NULL                                  ▷ The key was not found
    end if

    success ← TRYLOCKLRULISTMUTEX()
    if success then                ▷ Omit LRU node pull if unable to lock mutex
        listNode ← hashNode.listNode                ▷ Each hash table node has
                                                    ▷ a reference to its list node
        inList ← NODEISINLRULIST(listNode)
        if inList then     ▷ Node could have been evicted by another thread
            EVICTNODEFROMLIST(listNode)
            INSERTTOLRULISTHEAD(listNode)
        end if
        UNLOCKLRULISTMUTEX()
    end if

    return hashNode.value
end function
```

Algorithm 8 Insert element into HHVM cache

```
function Insert(key, value)
    listNode ← AllocateListNode()
    listNode.key ← key
    hashNode ← AllocateHashNode()
    hashNode.value ← value
    hashNode.listNode ← listNode

    success ← InsertIntoMap(key, hashNode)
    if not success then
        Delete(hashNode)
        Delete(listNode)
        return FALSE
    end if

    if GetCurrentItemCount() ≥ GetMaxCacheSize() then
        EvictNode()
        evictionDone ← TRUE
    else
        evictionDone ← FALSE
    end if

    LockLruListMutex()                ▷ Execution may be suspended here
    InsertToLruListHead(listNode)
    UnlockLruListMutex()

    if not evictionDone then
        IncreaseCurrentItemCount()
    end if

    return TRUE
end function
```

Chapter **4**

DeferredLRU Design

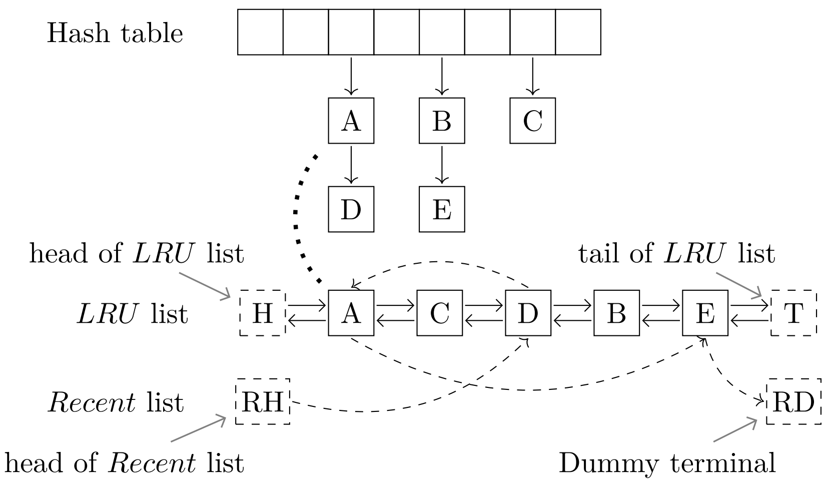
In this chapter, a novel concurrent SW cache, called **DeferredLRU**, is proposed. The reference implementation is available from its repository [54].

The main scalability-limiting factor for a concurrent LRU implementation (see Section 3.1 and Section 3.2) is the high contention happening on the *LRU* list head due to insertions performed on every Lookup and Insert.

DeferredLRU attempts to overcome the limitation. It avoids the contention by deferring insertions and performing them in carefully scheduled bulks. Accessed elements are stored in the so-called *Recent* list. DeferredLRU uses a concurrent hash table for a fast element lookup and a doubly linked *LRU* list for tracking access order.

An example of a DeferredLRU is depicted on Figure 4.1. It contains five elements (A, B, C, D, E). Element A is added to the *LRU* list most recently. Elements A, D, E are also added to the *Recent* list (element D is added last).

Figure 4.1: DeferredLRU structure



4.1 Data structures

4.1.1 Cache structure

DeferredLRU uses the following instance variables:

- total capacity of the SW cache
- current number of elements in the SW cache
- number of elements in the *Recent* list
- head element of the *LRU* list
- tail element of the *LRU* list
- link to the head element in the *Recent* list
- dummy terminal of the *Recent* list
- link to the head element of the empty element pool
- array of the hash table buckets
- atomic flags for requesting PullRecent
- atomic flag for requesting PurgeOld
- cache consolidation mutex (see Section 4.3.1)

4.1.2 Node structure

Each DeferredLRU node contains the following fields:

- *key*, *value* — user data
- *bucketNext* — link used to form buckets in a hash table
- *lruNext* — link to a next node in the *LRU* list
- *lruPrev* — link to a previous node in the *LRU* list
- *recentNext* — link to a next node in the *Recent* list

4.1.3 Initialization stage

The total cache capacity is known beforehand and is fixed. Therefore memory for all elements is preallocated during the initialization stage. DeferredLRU does not perform any dynamic allocations after the initialization stage.

4.1.4 Empty element pool

Initially all elements are empty. They are stored in an *empty element pool* — a singly linked list (reusing *bucketNext* link) of fresh nodes. When a new element is requested, the head element is taken from the empty element pool.

Empty element pool head is a point of contention as it is accessed on every insertion. Therefore, the insertion and removal routines should be scalable. This is achieved with lock-free lists. Specifically, DeferredLRU reuses lock-free list routines presented in [55] for managing empty element pool.

4.1.5 Hash table

DeferredLRU relies on a concurrent hash table with closed addressing as described in Section 2.3.1.1. It is represented as an array of buckets where each bucket is a head of a linked list. Each element key is mapped to its bucket.

Each hash table bucket has its own mutex. When a thread accesses a bucket, it must lock the corresponding bucket lock. This way, unsafe concurrent modifications of the bucket elements are avoided. What is more, bucket elements can be safely accessed and their content can be retrieved without any additional synchronization while a thread holds the bucket lock.

4.1.5.1 LRU list

DeferredLRU tracks element access order quite similarly to a typical LRU cache. All elements are made into a doubly linked list, new elements are added to the head, and the tail elements are considered least recently used. The difference is in the way the accessed elements are moved to the head. When an element is visited, the caller appends it to the *Recent* list (if the element has not been added already) instead of immediately pulling the element to the head.

4.1.6 Recent list

Recently accessed elements are added to the *Recent* list. It is a singly linked list. Elements use *recentNext* link to form this list. DeferredLRU keeps a link to the *Recent* list head (RH element on Figure 4.1) and the current number of elements in it. Initially, the *Recent* list head points to some dummy terminal (RD element on Figure 4.1). When the list is traversed, reaching this element terminates the traversing loop.

If an element is not in the *Recent* list, its *recentNext* link is set to NULL value. Later, when it is accessed, if the element is not yet in the *Recent* list, it is added to it. Thanks to the dummy terminal even the first added element has the link set to non-NULL value. With this approach, it is always possible to check if an element is in the *Recent* list by testing if the *recentNext* link is

Algorithm 9 Try insert node to *Recent* list

```
global $recentListHead$

procedure TryInsertToRecent($node$)
    if $node.recentNext \neq$ NULL then
        $next \leftarrow recentListHead$
        repeat
            $node.recentNext \leftarrow next$
        until AtomicCAS($recentListHead, next, node$)
    end if
end procedure
```

NULL or not. The insertion in the singly linked list is performed atomically with the AtomicCAS operation.

When the number of elements in the *Recent* list hits a certain threshold (for instance, 10% of the total capacity; see Section 5.2), the PullRecent operation is invoked. It extracts all elements in the *Recent* list from the *LRU* list and reinserts them in the head.

4.2 Cache operations

In this section, the implementation of the main cache operations — Lookup and Insert — is presented.

4.2.1 Lookup

The Lookup operation looks up a value by its key in the hash table. If found, it ensures that the accessed element is added to the *Recent* list (as defined in Section 4.1.6). If this causes the *Recent* list size to hit the threshold, the caller triggers PullRecent (see Section 4.3.1).

The presented algorithm has two interesting properties. Firstly, if the searched node is in *Recent* list, no updates to the *Recent* list or the *LRU* list has to be performed. Therefore, accessing recent elements becomes even faster.

With the current implementation, each Lookup has to lock the hash table which implies memory writes. However, there are more sophisticated hash tables that supports write-free lookups (see Section 2.3.1). Combining such hash table with DeferredLRU can improve cache scalability even more.

Secondly, accessing recent nodes does not increase *Recent* list size. Therefore, if the input sequence converges to a small subset of values and the whole subset fits into the *Recent* list, no costly PullRecent is performed.

Algorithm 10 Find element in cache

```
function FIND($key$, $result$)
    $bucket \leftarrow$ MAPKEYTOBUCKET($key$)
    LOCK($bucket$)                                  ▷ Lock corresponding mutex
    $found$, $node \leftarrow$ search $key$ in $bucket$ list
    if $found$ then
        $result \leftarrow node.value$
        TRYINSERTTORECENT($node$)
    end if
    UNLOCK($bucket$)

    if $Recent$ list list threshold is hit then
        TRIGGERPULL()                               ▷ See Algorithm 12
    end if

    return $found$
end function
```

4.2.2 Insert

The INSERT operation stores a new key-value pair in the cache. If the cache is full, it evicts some existing elements by triggering PURGEOLD. Then it initializes an empty element and inserts it into the cache.

However, the key could have been inserted already by another thread. Then the hash table insertion will fail and return False. In this case the allocated node is returned to the empty element pool.

The element is inserted first into the hash table and then into the *LRU* list (using the insertion described in Section 2.3.2). The reverse order would result in a race condition, when the element may be evicted from the *LRU* list before it has been inserted into its bucket.

However, doing the insertion in the presented order could also be flawed with the following race condition. Thread T_1 inserts an element X into its bucket. Thread T_2 accesses element X during LOOKUP and adds it to the *Recent* list. Thread T_3 starts PULLRECENT and tries to remove element X from the *LRU* list, but element X is not there yet.

To avoid this, the partially inserted element is prevented from being added to the *Recent* list by temporally assigning some non-NULL value to its *recentNext* link before inserting it to the cache. This way, the node can not be added to the *Recent* list (since it seems like it is already in the list). Also it can not be evicted neither with PULLRECENT (since it is not in the actual *Recent* list) nor with PURGEOLD (again, it seems like the element in the list already, thus it is skipped). After the insertion is complete, the *recentNext* link is reset to NULL allowing it to be processed normally.

Algorithm 11 Insert key-value pair into cache

```
global hashtable
global lruHead
global emptyPoolHead

procedure Insert(key, value)
    node ← AcquireEmptyNode()                          ▷ from node pool
    node.key, node.value ← key, value

    node.recentNext ← Recent list terminal
    inserted ← HTableInsert(hashtable, node)

    if inserted then
        DListAtomicInsert(lruHead, node, node)
        node.recentNext ← NULL
    else                              ▷ The same key is in hash table already
        DisposeNode(node)                        ▷ Return node to node pool
    end if
end procedure
```

4.3 Auxiliary operations

4.3.1 Cache consolidation

Cache consolidation is responsible for updating the cache structure to reflect recent element accesses. At most one thread is allowed to perform it at a time. The consolidations includes two optional stages:

- PullRecent resets and processes the *Recent* list. It is triggered when the size of the *Recent* list grows beyond the threshold.
- PurgeOld that evicts a portion of the least recently used elements from the *LRU* list and puts them in the empty element pool. It is triggered when a new element is attempted to be inserted, but the cache is full.

The cache consolidation routine is triggered when any of PullRecent and PurgeOld stages is triggered. The triggering is thread-safe and non-blocking. It is similar for both stages.

The triggering is performed in two steps. At first, an atomic binary flag (there is one for PullRecent and one for PurgeOld) is raised, signaling that the corresponding operation was requested when a consolidation is performed. Then the triggering thread attempts to perform the consolidation by itself. It tries to lock a mutex (using TryLock operation) that guards the consolidation. If locking is successful, the consolidation is performed.

Algorithm 12 Cache consolidation

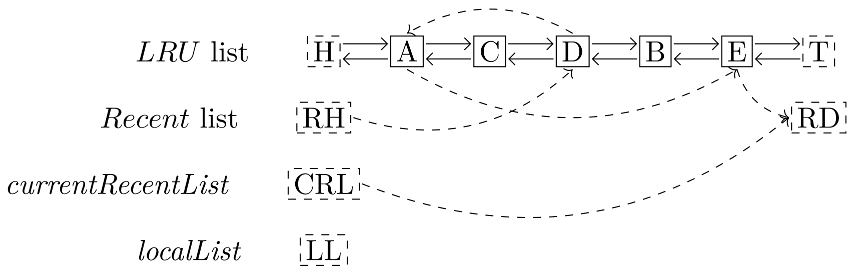
```
global pullFlag
global purgeFlag
global consolidationLock                                    ▷ Mutex

procedure TriggerPull()
    pullFlag ← TRUE
    Consolidate()
end procedure

procedure TriggerPurge()
    purgeFlag ← TRUE
    Consolidate()
end procedure

procedure Consolidate()
    if TryLock(consolidationLock) then
        if pullFlag = TRUE then
            PullRecent()
            pullFlag ← FALSE
        end if
        if purgeFlag = TRUE then
            PurgeOld(purgeCount)
            purgeFlag ← FALSE
        end if
        Unlock(consolidationLock)
    end if
end procedure
```

The opposite case means that another thread is already performing consolidation at the moment. In case of triggering PullRecent, no additional actions are required. It is not critical that nodes would be pulled later. From now every thread that adds a node to the *Recent* list attempts to trigger the consolidation. The first one to do so after the current consolidation is finished would perform the next consolidation cycle.

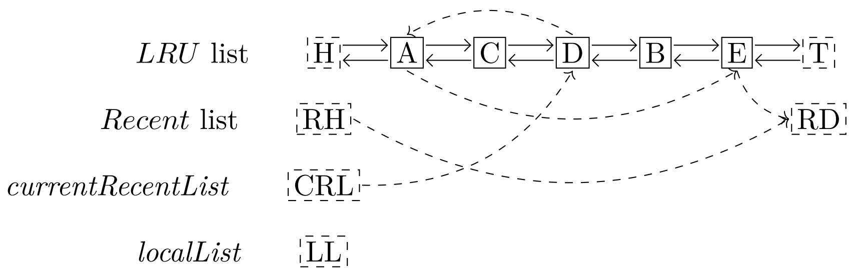
However, Insert cannot proceed without an empty element. Therefore, the thread would spin in a loop waiting for either the consolidating thread to put a new element to the empty element pool or the current consolidation to finish, so that it can perform another one by itself.

Figure 4.2: PullRecent execution

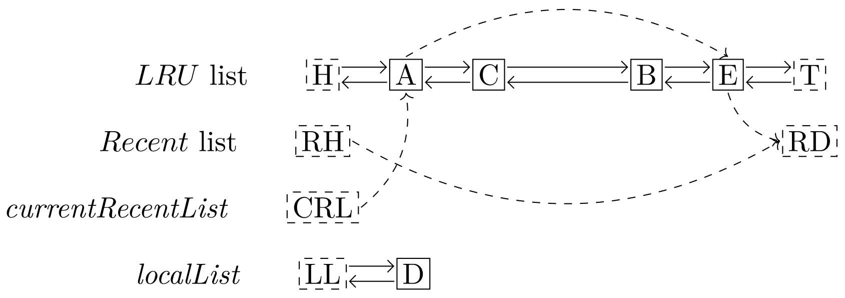
(a) Before PullRecent



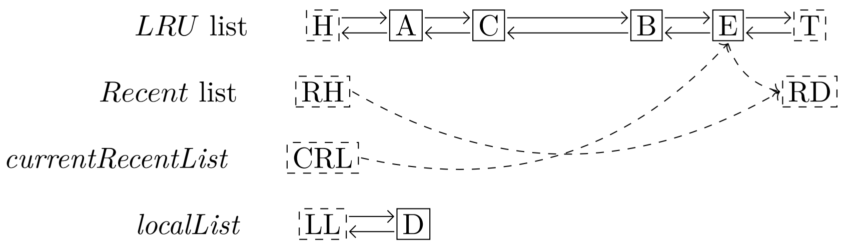
(b) Atomically exchange recent list head and *currentRecentList*



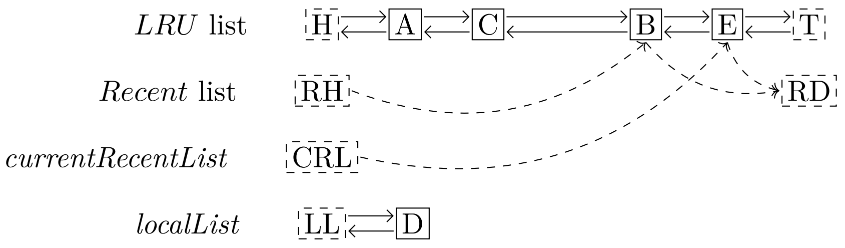
(c) Evict node D from *LRU* list



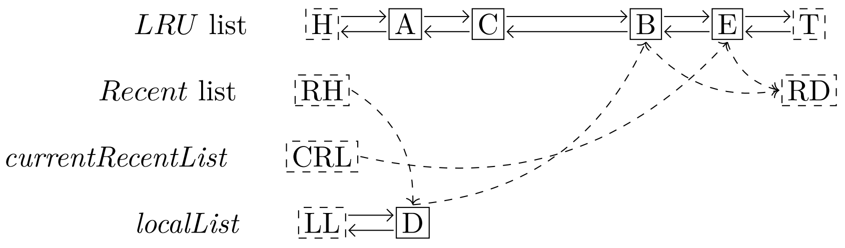
(d) Skip node A because it is near the *LRU* list head



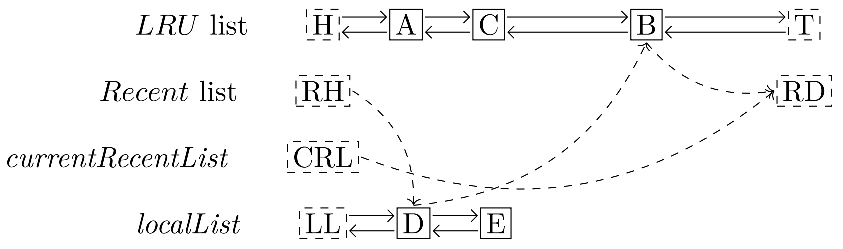
(e) Concurrently add node B to the *Recent* list



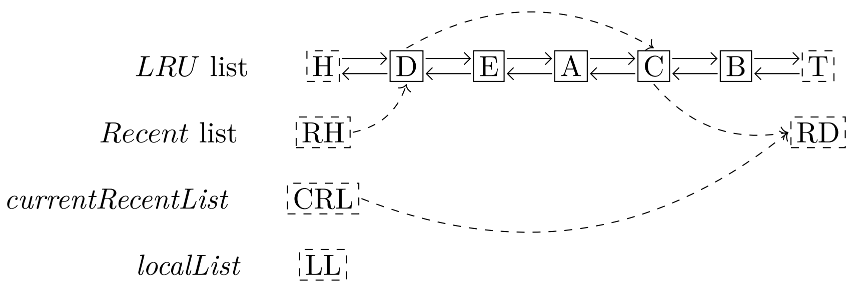
(f) Concurrently add node D to the *Recent* list



(g) Evict node E from *LRU* list



(h) Reinsert nodes in *localList* into the *LRU* list



Algorithm 13 Pull recently accessed node to front

```
global recentHead
global lruHead

procedure PullRecent()
    T ← Recent list list dummy terminal
    currentRecentList ← AtomicExch(recentHead, T)
    localList ← empty doubly linked list

    for all n ← elements in currentRecentList do
        if n.lruPrev ≠ lruHead then
            Evict n from LRU list list
            Insert n in the end of localList
        end if
        n.recentNext ← NULL
    end for

    first, last ← localList.first, localList.last
    DListAtomicInsert(listHead, first, last)
end procedure
```

4.3.2 PullRecent operation

PullRecent (Algorithm 13) is responsible for reordering accessed nodes in the *Recent* list. A step-by-step application of the algorithm is depicted in Figure 4.2.

PullRecent takes the slice of the *Recent* list and resets the list, as shown in Figure 4.2b. It does so atomically using the AtomicExch operation (see Section 2.2.3).

The slice is traversed by following the *recentNext* link until the *Recent* list dummy terminal (see Section 4.1.6) is reached. The nodes in the slice are removed from the *LRU* list and joined in a temporal list (see Figure 4.2c and Figure 4.2g). The *recentNext* links of the traversed nodes are reset.

Finally, this temporal list is inserted into the *LRU* list head (see Figure 4.2h) using the algorithm described in Section 2.3.2.

A node next to the *LRU* list head is skipped in order to satisfy the list insertion constraints, introduced in Section 4.1.5.1 (An element must not be removed if its *lruPrev* link points to the *LRU* list head). As shown in Figure 4.2d, node A is not removed from the list because it follows the head. The *recentNext* link of A is reset the same as for other nodes.

During PullRecent, other threads are still able to perform Lookup and Insert operations. The accessed nodes can be added to the *Recent* list as usual (see Figure 4.2e).

What is more, it is safe to add nodes that are currently being processed by PULLRECENT. As shown in Figure 4.2f, node D is added to the *Recent* list, while not being a part of the *LRU* list. This is correct because nodes may only be evicted during the cache consolidation process, and no other thread would start another consolidation until the current one is finished.

It is worth noting that all recent nodes are inserted in a single step. The total number of head insertions is $N_{\text{Insert}} + N_{\text{PullRecent}}$, while in a regular LRU cache, it is $N_{\text{Insert}} + N_{\text{Lookup}}$. The upper bound for $N_{\text{PullRecent}}$ is shown in Equation 4.1. The reduction of head insertions is the main reason for improved DeferredLRU scalability.

Each LOOKUP either increases the *Recent* list size by one (when the searched node is not in the *Recent* list yet) or does not change it. *Recent* list capacity — $C_{\text{Recent list}}$ — is usually set as a fraction of total cache capacity, thus $C_{\text{Recent list}} \gg 1$. PULLRECENT is executed each time *Recent* list is full. This implies Equation 4.1. Note that accessing only the nodes that are in the *Recent* list results in no PULLRECENT performed at all.

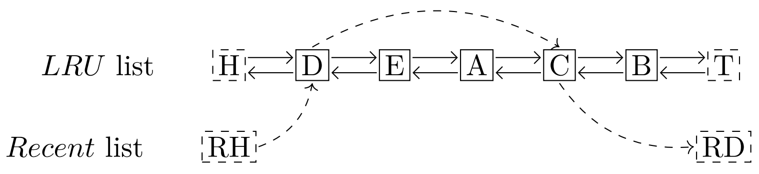
$$N_{\text{PullRecent}} \leq \frac{N_{\text{Lookup}}}{C_{\text{Recent list}}} \ll N_{\text{Lookup}} \tag{4.1}$$

4.3.3 PurgeOld operation

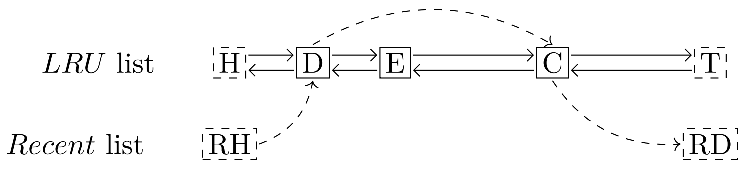
PURGEOLD evicts a number of the least recently used elements and puts them into the empty element pool so that they can be reused for further insertions. This is done by traversing the *LRU* list backward. Every element is removed first from the hash table and then from the *LRU* list (see Algorithm 14). However, some of these elements could have been accessed recently and, therefore, added to the *Recent* list. These elements are skipped.

Figure 4.3: PURGEOLD execution

(a) Before PURGEOLD



(b) Nodes A and B are evicted after PURGEOLD



Algorithm 14 Evict least recently accessed nodes to the empty node pool

```
global emptyPoolHead
global lruTail

procedure PurgeOld(targetCount)
    evictedCount ← 0
    node ← lruTail.lruPrev
    if node = lruHead then
        return                                    ▷ List is empty
    end if

    while evictedCount < targetCount do
        next ← node.lruPrev
        if next = lruHead then
            return                    ▷ Traversed complete LRU list
        end if

        removed ← HTableRemoveNonrecent(node)
        if removed then
            Evict node from LRU list
            Delete node key and value DisposeNode(node)
            evictedCount ← evictedCount + 1
        end if

        node ← next
    end while
end procedure
```

Figure 4.3 demonstrates the execution of PurgeOld. The target count is set to 2. The first node before the *LRU* list tail node — B — is evicted first. Although node C is the second node from the end, it is skipped as it is a part of the *Recent* list. As a result, nodes A and B are evicted.

The detection is of the recent nodes is done in two steps. At first, the *recentNext* link is checked to determine if the element is in the *Recent* list. Then the corresponding bucket is locked, and the element is searched in it to be removed. When found, its link is rechecked as it could have been changed between the first check and the bucket being locked (it cannot be changed after the bucket is locked). If the element has just been accessed, it is skipped, and the traversal goes on.

Chapter 5

Performance Evaluation

DeferredLRU is compared to existing alternatives using a benchmark program, that is capable of evaluating SW caches under different workloads and with varying number of threads. The performance evaluation focuses mainly on these aspects: throughput, hit-rate, and scalability.

Throughput corresponds to the number of operations made in unit of time. It is measured in millions of operations per second (MOp/s).

Hit-rate is a metric for evaluating caching efficiency. It is defined as a proportion of successful key lookups (cache hits) to all lookups.

Scalability tells how well cache maintains its throughput with rising number of threads. Scalability is evaluated as an increase of throughput with many threads compared to a single thread. This increase is denoted as *speedup* in this chapter.

This chapter proceeds as follows. Firstly, the test environment is described. Then the influence of DeferredLRU meta-parameters on its performance is analyzed. After this the results of the performance evaluation are presented. Finally, DeferredLRU is evaluated by integrating it in the actual application LSU3shell [18, 56] that heavily relies on the relevant SW caching.

5.1 Test setup

The benchmark application [53] allocates a cache with a given capacity and starts the requested number of threads. Each thread queries the cache with a sequence of keys. All results presented in this chapter have been collected without cache warmup.

The benchmark was done on the following machine: Intel® Xeon® Skylake CPU (16 HW cores @2.00GHz), 28 GB RAM, Ubuntu 18.04, GCC 7.3.0.

5.1.1 Data traces

For simulating realistic usage conditions, the sequences of cache requests are defined with data traces. These traces represent either memory accesses or network accesses of some real-world system. They have been used for evaluating state-of-art caching algorithms [22].

In addition, a synthetically generated data trace with Zipfian distribution [57] is used. [58] shows that a series of web requests from a fixed user community tends to have a Zipfian distribution. Since web requests caching is one of the major SW cache applications, this trace is relevant for cache performance evaluation.

The following traces are used:

DS1 [59] — disk operations of a database server running an ERP application. It has been used in [22, 23]. Available from [60].

OLTP — accesses to a CODASYL database. It has been used in [6, 19, 22, 23, 33]. Available from [60].

P4, P8 [61] — disk operations of Windows NT workstations. It has been used in [22, 23]. Available from [60].

S3 [22] — disk read accesses recorded on a search engine serving web search requests. It has been used in [22, 23]. Available from [60].

Wiki [62] — “a trace of 10% of all user requests issued to Wikipedia (in all languages)” [62] during the period between September and October 2007. It has been used in [23]. Available from [63].

YouTube — “a collection of traces from a campus network measurement on YouTube traffic. This collection contains trace data about user requests for specific YouTube content” [64]. It has been used in [23]. Available from [64].

Zipf — generated trace. Key frequency distribution follows Zipfian distribution [57] with parameter $\alpha = 0.9$.

Table 5.1 gives an insight on statistical distribution of keys in the traces.

Each trace test is named with a trace name and a fraction that denotes the capacity of the cache relatively to the count of unique keys in the trace, e.g. Wikipedia 1/10 means that cache capacity equals to 7913592/10 = 791359. For most tests, fractions 1/10 and 1/1000 used, but for the smaller traces the capacity has been capped at 4096 items.

Table 5.1: Data traces for performance evaluation

Trace	N	K	N/K	M_Δ	μ_Δ	σ_Δ	RSD_Δ
DS1	43704979	10516352	4.16	9002656	8123029	4450575	0.55
OLTP	914145	186880	4.89	2262	35218	90727	2.58
P4	19776090	5146832	3.84	371407	1307707	2289736	1.75
P8	42243785	977545	43.21	127948	441308	1461535	3.31
S3	16407702	1689882	9.71	837178	1259440	1386920	1.10
Wikipedia	86748777	7913592	10.96	2034	1508350	5651933	3.75
YouTube	1463644	493121	2.97	11728	99979	193047	1.93
Zipf	1000000	91356	10.95	4732	48913	105768	2.16

[1] N — total number of queries
[2] K — number of unique keys
[3] Δ — series of repetition intervals (that is, given key K, number of other keys between two consecutive occurrences of K)
[4] M_Δ — median of repetition intervals
[5] μ_Δ — average repetition interval
[6] σ_Δ — standard deviation of repetition interval
[7] $RSD_\Delta := \frac{\sigma_\Delta}{\mu_\Delta}$ — relative standard deviation of repetition interval

5.1.2 Evaluated SW caches

The benchmark compares the performance of the following LRU SW caches (the abbreviations given in parentheses denote short cache aliases):

DeferredLRU (DLRU) — the concurrent cache, presented in this paper.

Binned DeferredLRU (B-DLRU) — a binned version of DLRU.

LRU (LRU) — a basic implementation of a LRU cache, based on a hash table and a doubly linked list. It relies on a coarse-grained synchronization as described in Section 3.1.

Binned LRU (B-LRU) — a binned version of LRU.

Concurrent LRU (conLRU) — LRU cache with a fine-grained synchronization as described in Section 3.2. The scalability is limited by the thread contention on the *LRU* list head.

Binned Concurrent LRU (B-conLRU) — a binned version of conLRU.

TBB LRU (TBB) [28] — concurrent LRU cache from Intel TBB library (see Section 3.3).

HHVM LRU (HHVM) [29] — concurrent LRU cache from Facebook HHVM project (see Section 3.4). It uses binning internally.

Since binning (see Section 2.2.6) is a general approach, that is compatible with any cache data structure, it is not evaluated as a separate cache, but rather several implementations (including DeferredLRU) are combined with binning and evaluated once again. Containers that are not binned, are called singular in this section. Binning vastly improves cache scalability, therefore binned caches are compared separately.

5.2 Meta-parameter choice

DeferredLRU depends on two meta-parameters that can be tuned for better performance. These are *pull threshold* and *purge threshold*.

The pull threshold determines the maximum size of the *Recent* list. A higher value results in fewer costly PullRecent operations. On the other hand, with smaller values, node ordering is closer to the exact LRU order.

The purge threshold determines how many nodes are evicted during PurgeOld. The larger it is, the more potentially useful entries are evicted. But this also means that time-consuming PurgeOld is called less often, and consecutive insertions are also faster (because there are some free nodes in the empty element pool).

In order to find out how these parameters affect the DeferredLRU performance, the cache has been evaluated with a different combination of pull and purge thresholds (each ranging from 0.001 to 0.9). The evaluation has been done with 1 and 32 threads and with the following traces: Wikipedia 1/10, Wikipedia 1/1000, P4 1/10, P4 1/1000, P8 1/10, P8 1/238.

These traces were selected because they represent different key distributions. Based on Table 5.1, P4 has few repetitions per key (low N/K value), while P8 shows the opposite property — it has most repetitions per key among all used traces. The Wikipedia trace has an average N/K value, but it contains many local bursts of same key occurrences — M_Δ for this trace is 2034, while for P4 and P8, this value is 371407 and 127948 respectively.

The notation $a|b$ (where a and b are the pull threshold and the purge threshold, respectively) is used thereafter to denote a meta-parameter combination.

Heatmaps are chosen for presenting the enormous amount of the gathered data: Figure 5.1 and Figure 5.2 display throughput and hit-rate values for each combination of trace and thread count. Red and green colors denote lower and higher values, respectively. Appendix A provides all measured data in textual form.

5.2.1 Throughput evaluation

Throughput measurements show consistently low results in the lower-left area of the plots — around 0.4|0.001 and 0.9|0.01. All the peak values are in the area over the main diagonal.

With the single-thread measurements, the highest throughput is achieved with parameters set between 0.001|0.9 and 0.4|0.9 for the 5 of 6 traces. However, P8 1/10 peaks at 0.001|0.001.

With 32 threads, the overall trend is the same. The area with a purge threshold greater than 0.4 is optimal, while the best pull threshold setting is different for each trace. P8 1/10 peak is also shifted towards higher purge threshold values compared to the single-threaded results. On the other hand, the P4 1/10 peak is shifted towards 0.001|0.001 in contradistinction to the other traces.

5.2.2 Hit-rate evaluation

There is a clear trend in the hit-rate results. The highest value is achieved with a high pull threshold and a low purge threshold. Oddly enough, this is the area with the lowest throughput. Therefore, setting DeferredLRU parameters is a trade-off between the overall cache throughput and the hit-rate. For instance, for the single-threaded P8 1/238 trace test, the hit-rate plot appears to be an almost exact inversion of the throughput plot.

In the single-thread measurements, the pull and purge thresholds have a comparable effect on the resulting hit-rate. P4 1/10, P4 1/1000, and P8 1/10 depend more on the purge threshold being less than 0.4, while Wikipedia 1/1000 and P8 1/238 hit-rates are mostly determined by the pull threshold.

However, with the parallel evaluation, the pull threshold becomes the main factor for the hit-rate in 5 of 6 tests. P4 1/10 is the only trace in which the hit-rate is mainly influenced by the purge threshold in both tests. It is also the only trace that yields high throughput with lower purge throughput values in the 32 threads test.

5.2.3 Conclusion

General trends in throughput and hit-rate data are inverted to some extent. The following reasoning is based on the parallel evaluation, which is the primary use case for DeferredLRU.

The hit-rate mostly peaks around 0.9|0.01 and degrades rapidly with lower pull threshold values. Except for the P4 1/10 trace, the hit-rate depends weakly on the purge threshold. In most cases, the throughput is optimal with the pull threshold less than 0.4 or the purge threshold greater than 0.1.

In general, both throughput and hit-rate greatly depend on selected meta-parameter values. An optimal setting can be found empirically. DeferredLRU should be evaluated with different combinations of the thresholds, and the most suitable one should be used.

Figure 5.1: DeferredLRU throughput with different pull and purge thresholds

(a) 1 thread (b) 32 threads

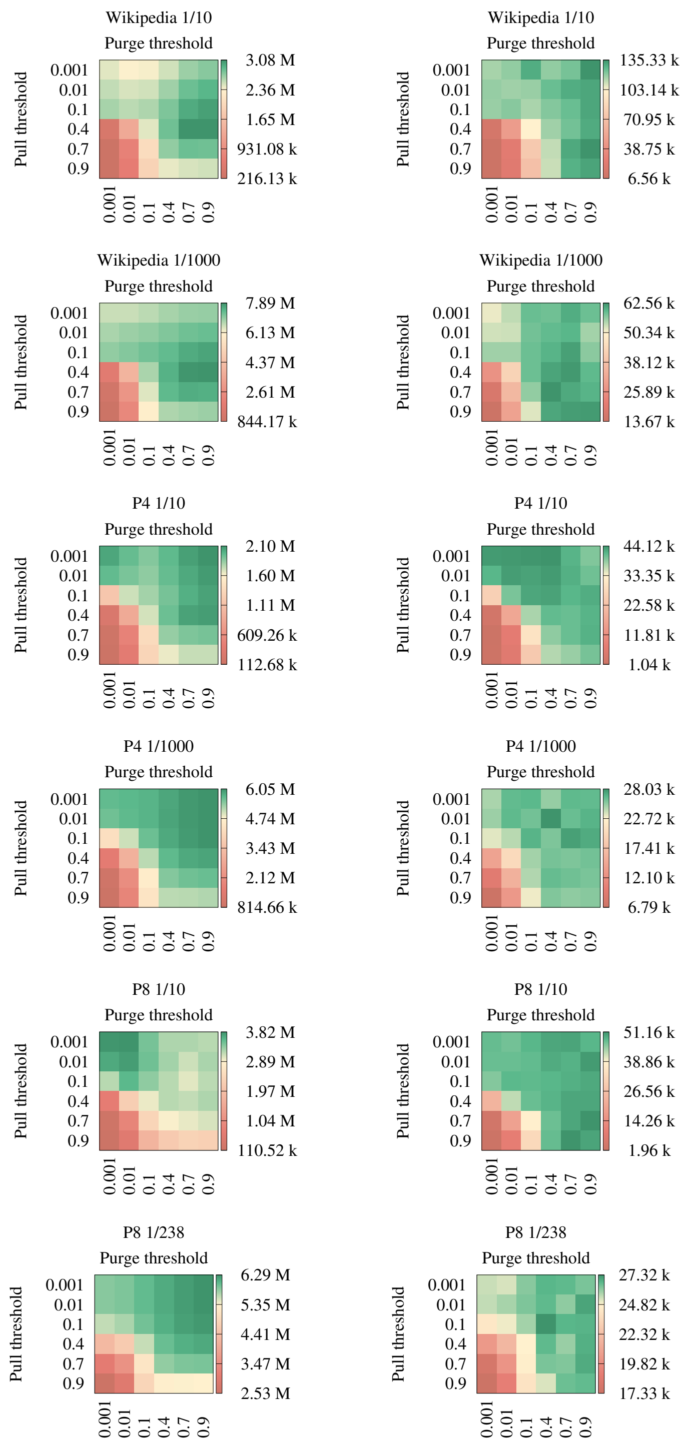
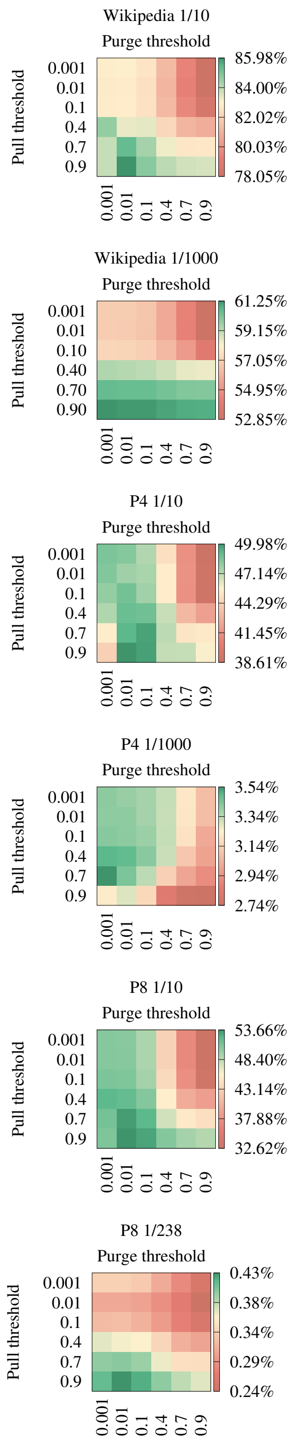
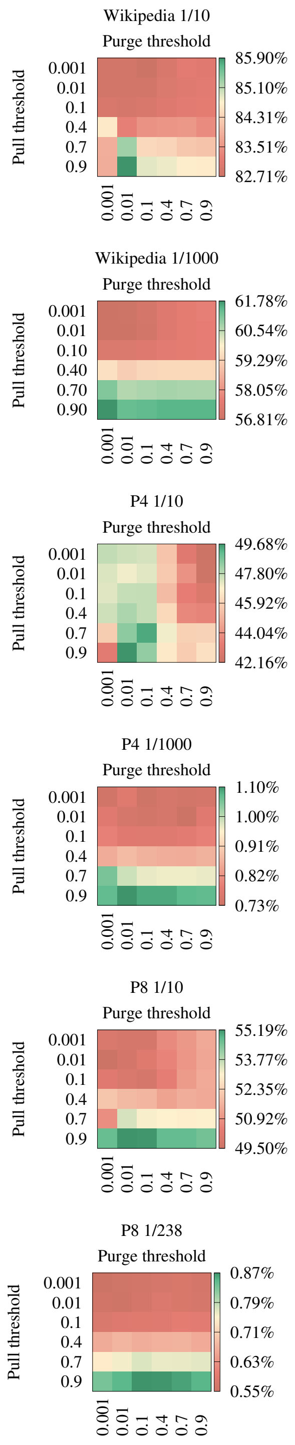


Figure 5.2: DeferredLRU hit-rate with different pull and purge thresholds

(a) 1 thread



(b) 32 threads



5.3 Hit-rate

When PullRecent is performed, items in the *Recent* list are reinserted into the *LRU* list not in the exact LRU order, but rather in the order of adding them into the *Recent* list list after the previous PullRecent. This order may potentially result in a different cache hit-rate.

In order to assert that DeferredLRU performs as well as a regular LRU cache, the best-achieved hit-rate for each test from Section 5.2 is compared to the hit-rates achieved by LRU and HHVM.

The results are presented in Table 5.2. DeferredLRU replacement strategy achieves *better hit-rate than a regular LRU scheme* in every test, exceeding the former by up to 7.8% (relatively). However, to achieve such superiority, the cache has to be fine-tuned for a particular input.

In some tests (for instance, P4 1/1000), hit-rate differs significantly for 1 and 32 threads. This happens because, in multi-threaded tests, worker threads replay the trace independently and at a different pace. Therefore, the cache is queried with different trace subsequences at the same time. This mixes up and nullifies local trace anomalies that are effective in a single-threaded test, like a short burst of occurrences of some key or scanning sequences (when many consequent keys are requested once).

Table 5.2: Hit-rate comparison

		Cache		
Trace	**Threads**	DeferredLRU	LRU	HHVM
Wikipedia 1/10	1	**85.98%**	82.72%	82.84%
	32	**85.90%**	83.77%	82.18%
Wikipedia 1/1000	1	**61.25%**	56.87%	56.91%
	32	**61.78%**	57.10%	56.36%
P4 1/10	1	**49.98%**	48.29%	48.17%
	32	**49.68%**	43.46%	47.87%
P4 1/1000	1	**3.54%**	3.41%	3.41%
	32	**1.10%**	0.74%	0.62%
P8 1/10	1	**53.66%**	50.63%	50.37%
	32	**55.19%**	49.17%	45.98%
P4 1/238	1	**0.43%**	0.34%	0.32%
	32	**0.87%**	0.65%	0.69%

5.4 Performance

The results of the primary performance evaluation are discussed in this section. Complete benchmark data is available in Appendix B. Tables 5.3 and 5.4 present data for the subset of traces for better perception.

The concurrent SW caches presented in Section 5.1.2 are evaluated with each data trace and a variable number of threads. For each run, throughput and hit-rate values are recorded. In order to obtain more precise data, each run is repeated 3 times, and then the results are averaged.

The following DeferredLRU meta-parameter settings are considered based on the evaluation in Section 5.2: DeferredLRU 0.001|0.1 performs well for traces similar to P4 1/10; DeferredLRU 0.1|0.7 is threshold optimal in most tests; DeferredLRU 0.99|0.99 is found to be competitive in both throughput and hit-rate tests. The setting 0.99|0.99 is chosen over 0.9|0.9 basing on another series of measurements that were accidentally lost during the experimentation.

5.4.1 Singular caches

In this section, benchmark measurements for the singular caches are analyzed. An excerpt of the singular measurements is presented in Table 5.3.

5.4.1.1 Hit-rate

In the single-threaded evaluation, DeferredLRU shows greater or equal hit-rate than the other caches. In 9 of 16 tests, at least one of the DeferredLRU configurations achieves comparable results with other caches; the difference is within 2%.

However, DeferredLRU 0.99|0.99 performs significantly better in some other tests. It achieves up to 4.3 times higher (S3 1/10 test) hit-rate than all other tested cached in 7 of 16 tests (DS1 1/10, P8 1/238, S3 1/10, S3 1/412, Wikipedia 1/1000, Zipf 1/10, and Zipf 1/22). Due to some data dependencies, the non-trivial DeferredLRU eviction order happens to catch more useful elements in these traces. At the same time, DeferredLRU 0.99|0.99 demonstrates very low hit-rate in some other tests (it has almost 2 times lower hit-rate in the P4 1/1000 test). An investigation of the nature of these anomalies is a part of future research.

With 32 threads, the hit-rate trend is similar, although abnormally high hit-rate of DeferredLRU is achieved in 8 of 16 tests. When comparing the difference between 1 and 32-threaded tests for each cache separately, DeferredLRU hit-rates change more favorably than the hit-rate of the other caches for the higher number of threads. Going from 1 to 32 threads, DeferredLRU achieves higher, same, and lower hit-rates in 8, 2, and 6 tests, respectively. For other caches, these counts are 4, 5, and 7, respectively.

5.4.1.2 Throughput

In single-threaded tests, the simple LRU cache shows the highest throughput most of the time. ConcurrentLRU and TBB LRU perform all the same operations that LRU does plus additional synchronization overhead. As a result, LRU is up to 3 times faster than ConcurrentLRU, and up to 4 times faster than TBB LRU. ConcurrentLRU outperforms TBB LRU in 13 of 16 tests.

In every test, there is at least one of the DeferredLRU configurations that is faster than ConcurrentLRU and TBB LRU. In the DS1 1/10 and Wikipedia 1/10 tests, DeferredLRU even outperforms LRU despite more complex insertion and lookup routines. The possible reason for this is the fact that DeferredLRU performs fewer memory-writes for recently accessed items (see Section 4.2.1).

DeferredLRU is the fastest singular SW cache in all parallel tests. It is the only singular cache that achieves higher throughput with multiple threads (in P4 1/10, P8 1/10, Wikipedia 1/10, and Zipf 1/10 tests).

Of the evaluated DeferredLRU configurations, DeferredLRU 0.001|0.1 is the fastest one in 13 of 16 single-threaded tests. DeferredLRU 0.99|0.99 is the slowest one in all tests except DS1 1/10, where it achieves the highest throughput, even outperforming the simple LRU cache.

In the parallel evaluation, the trend is different. Each configuration outperforms the others in some tests. DeferredLRU 0.001|0.1, DeferredLRU 0.1|0.7, and DeferredLRU 0.99|0.99 achieve the highest throughput values in 5, 4, and 7 tests, respectively. This supports the conclusion of Section 5.2 that the optimal choice of metaparameters depends on the input data.

5.4.2 Binned caches

In this section, benchmark measurements for the binned caches are analyzed. An excerpt of the binned measurements is presented in Table 5.4.

It is worth noting that with binning, the DeferredLRU thresholds are scaled by the binning factor that is 64 for this evaluation. For instance, the effective configuration for DeferredLRU 0.99|0.99 is 0.015|0.015. This is the reason for the uniformity in hit-rate and throughput results between different DeferredLRU configurations.

5.4.2.1 Hit-rate

Hit-rate values are more consistent in the binned evaluation between different caches. In most tests, the difference between DeferredLRU and other caches hit-rate is not more than 5% (relatively).

In OLTP 1/45, P8 1/10, and both Zipf tests, DeferredLRU achieves lower hit-rate than the other caches. This is consistent with the performance of DeferredLRU 0.001|0.1 in the singular evaluation. On the other hand, in DS1 1/10 and DS1 1/1000 tests, DeferredLRU shows significantly higher hit-rate.

5.4.2.2 Throughput

In the single-threaded evaluation, LRU is faster than the other caches in 15 of 16 cases due to less synchronization overhead. In these tests, DeferredLRU consistently achieves 2/3 of the LRU performance.

HHVM achieves the highest throughput in the Wikipedia 1/10 test. At the same time, it scales worse than the other caches on this workload and shows one of the lowest throughputs with 32 threads.

Due to better scalability, DeferredLRU achieves significantly higher throughput in the parallel tests compared to the single-threaded evaluation. It outperforms all other caches in 11 of 16 tests. In the Zipf 1/10 test, DeferredLRU performs about 2.8 times faster than other caches. The supremacy is on a par with the singular evaluation, where DeferredLRU is 2.5 times faster than LRU.

In DS1 1/10, DS1 1/1000, P4 1/1000, P8 1/238, and YouTube 1/120, DeferredLRU is slower than LRU by up to 20%. However, DeferredLRU achieves a much higher speedup in these tests. This hints that DeferredLRU performs faster than LRU with even more threads.

For instance, in the DS1 1/10 test, the speedup between 16 and 32 threads is 1.76x for DeferredLRU and 1.38x for LRU. ConcurrentLRU achieves the highest 1-to-32-threads speedup in this test. It is slightly higher than the speedup of DeferredLRU (23.2x and 22.1x, respectively). However, its throughput is about 25% less than what DeferredLRU achieves. Most probably, ConcurrentLRU is not able to compete with DeferredLRU with a higher number of threads.

Figure 5.3 and Figure 5.4 show how throughput changes with an increasing number of threads for different SW caches. It is to see that DeferredLRU scales more steadily than other caches.

5.5 Conclusion

The evaluation presented in this chapter supports the statement that DeferredLRU scales better than the existing alternatives while being able to deliver comparable hit-rate and single-threaded performance.

Table 5.3: Singular cache performance

		Hit-rate		**Throughput**			
		Threads		Threads			
Trace	**Cache**	1	32	1	16	32	**Speedup**
DS1 1/10	DLRU 0.001|0.1	3.25%	4.90%	1.95	0.89	**0.85**	**0.44**
	DLRU 0.1|0.7	3.32%	5.52%	2.32	0.88	0.84	0.36
	DLRU 0.99|0.99	**6.73%**	**12.00%**	**3.62**	**0.90**	0.84	0.23
	LRU	3.43%	5.58%	3.07	0.73	0.67	0.22
	conLRU	3.55%	3.64%	1.56	0.66	0.50	0.32
	TBB	3.44%	3.51%	2.09	0.48	0.30	0.14
OLTP 1/45	DLRU 0.001|0.1	50.63%	32.51%	6.59	1.28	1.20	0.18
	DLRU 0.1|0.7	46.00%	33.33%	7.26	1.31	1.24	0.17
	DLRU 0.99|0.99	46.44%	41.99%	5.44	**1.41**	**1.40**	**0.26**
	LRU	51.27%	32.58%	**10.66**	0.94	0.82	0.08
	conLRU	**51.27%**	34.63%	5.42	0.75	0.56	0.10
	TBB	51.27%	**43.45%**	2.93	0.84	0.52	0.18
P4 1/10	DLRU 0.001|0.1	47.55%	**47.43%**	1.84	**1.53**	**1.43**	0.78
	DLRU 0.1|0.7	41.66%	42.73%	1.95	1.39	1.32	0.68
	DLRU 0.99|0.99	45.75%	44.83%	0.80	1.35	1.29	**1.60**
	LRU	**48.38%**	42.84%	**2.97**	0.89	0.79	0.27
	conLRU	48.34%	46.68%	1.68	0.86	0.61	0.36
	TBB	48.16%	45.94%	1.85	0.51	0.32	0.17
P8 1/10	DLRU 0.001|0.1	48.99%	49.84%	3.54	1.69	1.54	0.44
	DLRU 0.1|0.7	37.52%	50.93%	3.13	1.68	1.56	0.50
	DLRU 0.99|0.99	**51.06%**	**56.10%**	1.04	**1.78**	**1.67**	**1.61**
	LRU	50.62%	48.61%	**8.45**	1.08	0.93	0.11
	conLRU	50.59%	51.01%	3.11	0.87	0.63	0.20
	TBB	50.43%	50.34%	1.95	0.61	0.38	0.20
Wikipedia 1/10	DLRU 0.001|0.1	82.72%	82.64%	2.24	5.24	3.89	1.74
	DLRU 0.1|0.7	79.72%	83.09%	**2.77**	**5.82**	4.22	1.52
	DLRU 0.99|0.99	**83.92%**	**85.10%**	1.43	5.13	**4.30**	**3.01**
	LRU	82.92%	82.91%	2.40	0.90	0.78	0.33
	conLRU	82.47%	83.03%	1.52	0.98	0.75	0.49
	TBB	83.81%	84.12%	0.83	0.43	0.29	0.35
Zipf 1/10	DLRU 0.001|0.1	58.55%	59.40%	7.39	2.01	1.97	0.27
	DLRU 0.1|0.7	54.14%	60.28%	7.79	2.13	1.98	0.25
	DLRU 0.99|0.99	**66.86%**	**65.03%**	1.83	**2.41**	**2.21**	**1.20**
	LRU	59.26%	59.22%	**9.88**	0.97	0.88	0.09
	conLRU	59.26%	59.20%	5.35	0.89	0.69	0.13
	TBB	59.26%	59.25%	2.39	0.87	0.53	0.22

Table 5.4: Binned cache performance

		Hit-rate		**Throughput**			
		Threads		Threads			
Trace	**Cache**	1	32	1	16	32	**Speedup**
DS1	B-DLRU 0.001|0.1	**3.95%**	**8.36%**	1.00	12.45	21.83	21.75
1/10	B-DLRU 0.1|0.7	3.95%	8.31%	1.00	12.54	22.05	22.07
	B-DLRU 0.99|0.99	3.93%	8.10%	1.00	12.48	21.86	21.79
	B-LRU	3.63%	6.87%	**1.53**	**19.09**	**26.32**	17.20
	B-conLRU	3.33%	5.91%	0.69	10.97	16.11	**23.23**
	HHVM	3.41%	5.76%	1.10	8.91	10.53	9.55
OLTP	B-DLRU 0.001|0.1	48.49%	32.27%	6.16	26.82	**37.10**	**6.02**
1/45	B-DLRU 0.1|0.7	48.49%	32.36%	6.17	26.83	36.92	5.98
	B-DLRU 0.99|0.99	48.49%	32.35%	6.17	26.71	36.69	5.95
	B-LRU	51.13%	**34.79%**	**9.42**	**27.11**	34.45	3.66
	B-conLRU	51.14%	33.32%	5.12	16.47	20.30	3.96
	HHVM	**51.22%**	32.78%	4.75	16.11	17.20	3.62
P4	B-DLRU 0.001|0.1	47.98%	46.67%	1.37	23.38	39.35	28.77
1/10	B-DLRU 0.1|0.7	47.98%	46.79%	1.37	23.67	**39.35**	**28.82**
	B-DLRU 0.99|0.99	**48.21%**	46.63%	1.38	23.68	39.27	28.52
	B-LRU	48.04%	36.65%	**2.14**	**26.66**	32.54	15.17
	B-conLRU	48.20%	46.26%	1.07	17.34	22.67	21.09
	HHVM	47.88%	**47.90%**	1.47	12.72	15.61	10.58
P8	B-DLRU 0.001|0.1	49.19%	44.41%	3.19	31.30	43.96	13.79
1/10	B-DLRU 0.1|0.7	49.20%	44.43%	3.18	30.98	**44.06**	13.85
	B-DLRU 0.99|0.99	49.19%	44.40%	3.17	30.96	44.02	**13.88**
	B-LRU	**50.58%**	46.19%	**6.08**	**36.04**	41.07	6.75
	B-conLRU	50.49%	**46.28%**	2.62	22.13	26.77	10.22
	HHVM	50.40%	46.02%	1.94	16.97	18.60	9.58
Wikipedia	B-DLRU 0.001|0.1	82.71%	81.81%	2.38	35.71	55.12	23.12
1/10	B-DLRU 0.1|0.7	82.73%	81.80%	2.36	35.35	**57.04**	**24.18**
	B-DLRU 0.99|0.99	82.71%	81.78%	2.38	**35.72**	56.98	23.98
	B-LRU	**82.99%**	82.21%	2.33	24.02	31.75	13.63
	B-conLRU	82.44%	**82.39%**	1.46	18.19	24.45	16.69
	HHVM	82.92%	82.14%	**2.40**	20.06	25.53	10.65
Zipf	B-DLRU 0.001|0.1	57.86%	57.95%	6.94	**36.59**	51.12	**7.37**
1/10	B-DLRU 0.1|0.7	57.86%	57.94%	6.97	36.11	50.75	7.28
	B-DLRU 0.99|0.99	57.86%	57.95%	7.00	35.81	**51.24**	7.33
	B-LRU	59.22%	**59.18%**	**8.98**	25.38	18.20	2.03
	B-conLRU	59.22%	59.17%	5.16	9.22	4.24	0.82
	HHVM	**59.26%**	58.90%	4.74	18.84	21.72	4.58

Figure 5.3: Performance on Wikipedia 1/1000 trace

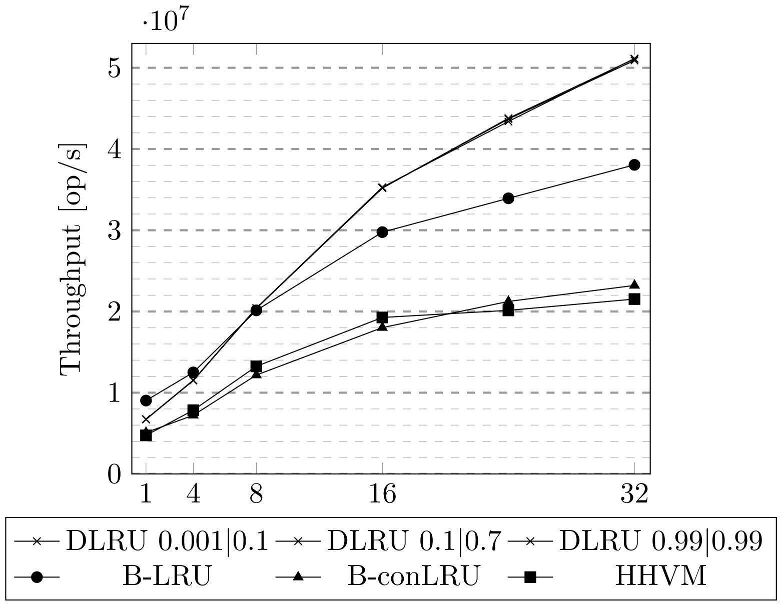
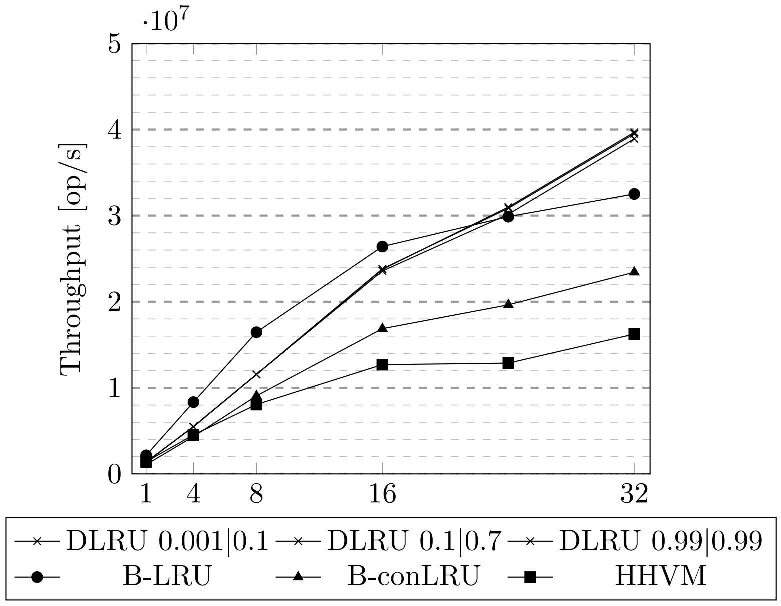


Figure 5.4: Performance on P4 1/1000 trace



5.6 LSU3shell

In addition to the synthetic evaluation, DeferredLRU performance was measured on the real application LSU3shell [18, 56] that heavily relies on memoization.

The LSU3shell is a highly scalable parallel code that implements an innovative many-body technique, dubbed symmetry-adapted no-core shell model (SA-NCSM), for modeling the structures of atomic nuclei from first principles. It solves the Schrödinger equation for a system of strongly interacting nucleons by casting it into the matrix eigenvalue problem.

The following caches are evaluated: Hash Table (as a baseline), DeferredLRU, Binned LRU, Binned Concurrent LRU, and Binned DeferredLRU. For DeferredLRU, pull and purge thresholds are set to 0.9 and 0.6 respectively. In this test the cache capacity has been set to be high enough to keep all elements at once. Hash Table serves as a cache without eviction strategy. It sets the lower bound on possible application running time.

LSU3shell relies on memoization of dynamically-sized values. A DeferredLRU extension that respects both item count and their cumulative size has been developed for this test, but it is impossible to do so with third-party caches without code rewriting. It is not possible to include HHVM LRU and TBB LRU caches for comparison.

Averaging results of multiple runs in Table 5.5 shows that LSU3shell with Binned DeferredLRU runs about 5% faster than its original implementation with a regular LRU cache and is almost as fast as the baseline.

Table 5.5: LSU3shell running time with different caches [s]

	Threads	
Cache	16	32
Hash table *(baseline)*	*1297*	*936*
DLRU	1430	986
B-DLRU	**1402**	**943**
B-LRU	1471	985
B-conLRU	1559	1088

Chapter **6**

Recommendations for Future Work

The presented study of DeferredLRU and its implementation leaves room for additional experiments and possible improvements. The following concerns may be addressed in future research.

6.1 Write-free cache lookup

Memory writes are generally more expensive than memory reads. A cache that allows write-free accesses to recent elements should perform much better than a regular one. The difference caused by the more efficient memory access may be more considerable in real complex applications than in benchmarks.

As mentioned in Section 4.2.1, DeferredLRU performs no memory writes in *LRU* list and *Recent* list for items that are in the *Recent* list already. However, with the current design, each access still involves hash table locking (that writes memory).

The sequential version of DeferredLRU requires no hash table synchronization. Therefore, it already provides write-free access to the recent items. The performance of this implementation should be evaluated against regular sequential caches.

The Hopscotch [50] and CLHT-LF [47] hash tables provide concurrent write-free Lookup operation. Embedding one of these hash tables into DeferredLRU may result a concurrent SW cache that performs no memory writes during lookup that for the recent items. It may yield a significant performance gain. To the best of the author's knowledge, it would be the first general-purpose concurrent SW cache with this property.

6.2 Improved eviction strategy

Although DeferredLRU eviction strategy is similar to LRU, it has slightly different properties that result in a superior hit-rate (see Section 5.3). The strategy should be studied more thoroughly, especially with respect to the meta parameter value.

The principle of deferring node removals can be used to build more sophisticated eviction strategies (such as those described in Section 1.1). Many of them are based on a variation of LRU cache that can be replaced with a scalable DeferredLRU implementation.

6.3 Improved hash table design

The implementation presented in Chapter 4 relies on a simple lock-based concurrent hash table. The possibility of using a more sophisticated, possibly lock-free design should be investigated. Such design may yield better scalability, although the cooperation with the concurrent *LRU* list may become more complex.

The hash table in DeferredLRU uses basic open hashing (see Section 2.1.3). This involves traversing a linked list of nodes on each hash table lookup, insertion, and removal. Since nodes are randomly distributed in memory, such design has a negative impact on the CPU cache. There are hash tables that show better memory access locality [36, 37, 46, 65, 66]. Using advanced hash table may improve both single-threaded and concurrent cache throughput.

6.4 Additional evaluation

DeferredLRU meta parameters setting has a considerable effect on the cache throughput and hit-rate. Furthermore, optimal values differ from trace to trace. There is no single best option for all cases. It should be investigated whether there is any direct dependence between the input data properties and the meta parameters effect and whether it is possible to adapt these parameters in runtime automatically.

Caffeine [33] is a concurrent TinyLFU-based cache implemented in Java. It provides a versatile multi-threaded benchmark that compares several state-of-art caches. With a DeferredLRU Java implementation, it would be possible to evaluate DeferredLRU using this benchmark and gain additional performance data.

Conclusion

A SW cache is used to omit repeating computations by storing values in memory. With a limited memory, it tries to remember only the most relevant elements. LRU is a simple strategy for choosing such elements.

Concurrent LRU implementations does not scale very well with multiple threads. DeferredLRU is a novel data structure for software caching that has better performance than LRU on many-thread systems thanks to a different approach to moving accessed elements. They are recorded in a helper list and moved in a batch when the list grows past some threshold. This trick lowers thread contention on the LRU list head.

The extensive performance evaluation of the DeferredLRU implementation and other concurrent caches is presented in Chapter 5.

At first, the reasoning on setting the DeferredLRU meta-parameters — pull threshold and purge threshold — is provided. It was found out that the optimal settings depend on input data. What is more, setting these meta-parameters provides a trade-off between cache throughput and hit-rate. In most tests, there was no single configuration that achieved both maximum throughput and hit-rate.

Then the results of the primary evaluation are discussed. The caches were evaluated with 16 different benchmark configurations and for 1, 16, and 32 threads. DeferredLRU achieved comparable and, in some tests, significantly higher hit-rate than LRU. While single-threaded throughput is lower than it is of LRU, with a higher number of threads, DeferredLRU achieved the highest performance of most tests.

Finally, DeferredLRU performance was evaluated by embedding it into LSU3shell parallel scientific application. With DeferredLRU, the execution time has been improved by about 5% compared to a simple LRU implementation.

Chapter 6 discusses the possible ways to improve DeferredLRU. The two main options are to use a more sophisticated hash table or to employ an advanced replacement strategy.

Bibliography

[1] Iyengar, A. Design and performance of a general-purpose software cache. *1999 IEEE International Performance, Computing and Communications Conference (Cat. No.99CH36305)*, 1999, doi:10.1109/pccc.1999.749456.

[2] Bilal, M.; Kang, S.-G. A Cache Management Scheme for Efficient Content Eviction and Replication in Cache Networks. *IEEE Access*, volume 5, 2017: p. 1692–1701, ISSN 2169-3536, doi:10.1109/access.2017.2669344. Available from: http://dx.doi.org/10.1109/ACCESS.2017.2669344

[3] Zhou, Y.; Philbin, J.; et al. The Multi-Queue Replacement Algorithm for Second Level Buffer Caches. In *Proceedings of the General Track: 2001 USENIX Annual Technical Conference*, USA: USENIX Association, 2001, ISBN 188044609X, p. 91–104, doi:10.5555/647055.715773.

[4] Benet, J. IPFS - Content Addressed, Versioned, P2P File System. *ArXiv*, volume abs/1407.3561, 2014. Available from: https://arxiv.org/abs/1407.3561

[5] OpenAFS: open-source implementation of the Andrew distributed file system. Accessed 29/12/2019. Available from: https://www.openafs.org/

[6] Johnson, T.; Shasha, D. E. 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. In *VLDB*, edited by J. B. Bocca; M. Jarke; C. Zaniolo, Morgan Kaufmann, 1994, ISBN 1-55860-153-8, pp. 439–450. Available from: http://dblp.uni-trier.de/db/conf/vldb/vldb94.html#JohnsonS94

[7] Dar, S.; Franklin, M. J.; et al. Semantic Data Caching and Replacement. In *Proceedings of the 22th International Conference on Very Large Data Bases*, VLDB '96, San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1996, ISBN 1558603824, p. 330–341, doi:10.5555/645922.673462.

[8] memcached – a distributed memory object caching system. Accessed 29/12/2019. Available from: https://memcached.org/

[9] Redis. Accessed 29/12/2019. Available from: https://redis.io/

[10] Nishtala, R.; Fugal, H.; et al. Scaling Memcache at Facebook. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, Lombard, IL: USENIX, 2013, ISBN 978-1-931971-00-3, pp. 385–398. Available from: https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/nishtala

[11] Wiger, N.; Timalsina, R. Performance at Scale with Amazon ElastiCache. *AWS*, 2018, accessed 29/12/2019. Available from: https://d0.awsstatic.com/whitepapers/performance-at-scale-with-amazon-elasticache.pdf

[12] Marques, R.; Swift, T.; et al. A Simple and Efficient Implementation of Concurrent Local Tabling. *Practical Aspects of Declarative Languages*, 2010: pp. 264–278, doi:10.1007/978-3-642-11503-5_22.

[13] Michie, D. Memo Functions and Machine Learning. *Nature*, volume 218, no. 5138, 1968: pp. 19–22, doi:10.1038/218306c0.

[14] Kirchner, H.; Levi, G. *Algebraic and logic programming*. Springer-Verlag, 1992, 128–142 pp.

[15] Mayfield, J.; Finin, T.; et al. Using Automatic Memoization As a Software Engineering Tool in Real-world AI Systems. In *Proceedings of the 11th Conference on Artificial Intelligence for Applications*, CAIA '95, Washington, DC, USA: IEEE Computer Society, 1995, ISBN 0-8186-7070-3, pp. 87–. Available from: http://dl.acm.org/citation.cfm?id=791219.791666

[16] Khalvati, F.; Aagaard, M. D.; et al. Window memoization: toward high-performance image processing software. *Journal of Real-Time Image Processing*, volume 10, no. 1, 2012: pp. 5–25, doi:10.1007/s11554-012-0247-8.

[17] Park, S.; Bahri, C.; et al. Numerical database system based on a weighted search tree. *Computer Physics Communications*, volume 82, no. 2-3, 1994: pp. 247–264, doi:10.1016/0010-4655(94)90172-4.

[18] Dytrych, T.; Maris, P.; et al. Efficacy of the SU(3) scheme for ab initio large-scale calculations beyond the lightest nuclei. *Computer Physics Communications*, volume 207, 2016: pp. 202–210, doi:10.1016/j.cpc.2016.06.006.

[19] O'Neil, E. J.; O'Neil, P. E.; et al. The LRU-K page replacement algorithm for database disk buffering. *ACM SIGMOD Record*, volume 22, no. 2, June 1993: pp. 297–306, ISSN 0163-5808, doi:10.1145/170036.170081. Available from: https://doi.org/10.1145/170036.170081

[20] Bélády, L. A. A Study of Replacement Algorithms for Virtual-Storage Computer. *IBM Systems Journal*, volume 5, 1966: pp. 78–101.

[21] Jiang, S.; Zhang, X. F. LIRS: An Efficient Low Inter-reference Recency Set Replacement to Improve Buffer Cache Performance. 1 2002, pp. 31–42, doi:10.1145/511399.511340.

[22] Megiddo, N.; Modha, D. S. ARC: A Self-Tuning, Low Overhead Replacement Cache. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, FAST '03, USA: USENIX Association, 2003, p. 115–130, doi:10.5555/1090694.1090708.

[23] Einziger, G.; Friedman, R.; et al. TinyLFU: A Highly Efficient Cache Admission Policy. *ACM Trans. Storage*, volume 13, no. 4, Nov. 2017, ISSN 1553-3077, doi:10.1145/3149371. Available from: https://doi.org/10.1145/3149371

[24] Bloom, B. H. Space/Time Trade-Offs in Hash Coding with Allowable Errors. *Commun. ACM*, volume 13, no. 7, July 1970: p. 422–426, ISSN 0001-0782, doi:10.1145/362686.362692. Available from: https://doi.org/10.1145/362686.362692

[25] Jain, A.; Lin, C. Back to the Future: Leveraging Belady's Algorithm for Improved Cache Replacement. In *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, IEEE Press, 2016, ISBN 9781467389471, p. 78–89, doi:10.1109/ISCA.2016.17. Available from: https://doi.org/10.1109/ISCA.2016.17

[26] 42 Years of Microprocessor Trend Data. Accessed 2/1/2020. Available from: https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/

[27] Moore, G. Cramming more components onto integrated circuits. *Solid-State Circuits Newsletter, IEEE*, volume 11, 10 2006: pp. 33–35, doi:10.1109/N-SSC.2006.4785860, reprinted from Electronics, volume 38, number 8, April 19, 1965, pp.114 ff.

[28] Intel Threading Building Blocks library. Accessed 29/12/2019. Available from: https://software.intel.com/en-us/intel-tbb

[29] HHVM – Virtual machine for Hack and PHP. Accessed 29/12/2019. Available from: https://github.com/facebook/hhvm

[30] BagLRU: A High Performance Multi-Threaded LRU Cache. Accessed 29/12/2019. Available from: https://www.codeproject.com/Articles/23396/A-High-Performance-Multi-Threaded-LRU-Cache

[31] eBay Tech Blog: High-Throughput, Thread-Safe, LRU Caching. Accessed 29/12/2019. Available from: https://tech.ebayinc.com/engineering/high-throughput-thread-safe-lru-caching/

[32] Guava: Google Core Libraries for Java. Accessed 29/12/2019. Available from: https://github.com/google/guava

[33] Caffeine: A high performance caching library for Java 8. Accessed 29/12/2019. Available from: https://github.com/ben-manes/caffeine

[34] Sedgewick, R.; Wayne, K. D. *Algorithms*. Addison-Wesley, fourth edition, 2011, ISBN 9780321573513.

[35] Knott, G. D. Hashing functions. *The Computer Journal*, volume 18, no. 3, 1 1975: pp. 265–278, ISSN 0010-4620, doi:10.1093/comjnl/18.3.265, http://oup.prod.sis.lan/comjnl/article-pdf/18/3/265/1138029/180265.pdf. Available from: https://doi.org/10.1093/comjnl/18.3.265

[36] Pagh, R.; Rodler, F. F. Cuckoo hashing. *Journal of Algorithms*, volume 51, no. 2, 2004: pp. 122–144, ISSN 0196-6774, doi:https://doi.org/10.1016/j.jalgor.2003.12.002. Available from: http://www.sciencedirect.com/science/article/pii/S0196677403001925

[37] Herlihy, M.; Shavit, N.; et al. Hopscotch Hashing. In *Proceedings of the 22nd International Symposium on Distributed Computing*, DISC '08, Berlin, Heidelberg: Springer-Verlag, 2008, ISBN 9783540877783, p. 350–364, doi:10.1007/978-3-540-87779-0_24. Available from: https://doi.org/10.1007/978-3-540-87779-0_24

[38] David, T. A.; Guerraoui, R.; et al. Designing ASCY-compliant Concurrent Search Data Structures. 2014: p. 23. Available from: http://infoscience.epfl.ch/record/203822

[39] "Murphy's Law", The Merriam-Webster.com Dictionary. Accessed 29/12/2019. Available from: https://www.merriam-webster.com/dictionary/Murphy%27s%20Law

[40] Intel® 64 and IA-32 Architectures Developer‘s Manual. Accessed 14/1/2020. Available from: https://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-vol-2b-manual.html

[41] Arm® Architecture Reference Manual Armv8. Accessed 14/1/2020. Available from: https://developer.arm.com/docs/ddi0487/latest

[42] Fraser, K. Practical lock-freedom. Technical report UCAM-CL-TR-579, Computer Laboratory, University of Cambridge, Feb. 2004.

[43] Dechev, D.; Pirkelbauer, P.; et al. Understanding and Effectively Preventing the ABA Problem in Descriptor-Based Lock-Free Designs. In *2010 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, May 2010, ISSN 2375-5261, pp. 185–192, doi:10.1109/ISORC.2010.10.

[44] Chen, Z.; He, X.; et al. Concurrent hash tables on multicore machines: Comparison, evaluation and implications. *Future Generation Computer Systems*, volume 82, 2018: pp. 127–141, ISSN 0167-739X, doi:https://doi.org/10.1016/j.future.2017.12.054. Available from: http://www.sciencedirect.com/science/article/pii/S0167739X17317715

[45] Maier, T.; Sanders, P.; et al. Concurrent Hash Tables: Fast and General(?)! *ACM Trans. Parallel Comput.*, volume 5, no. 4, Feb. 2019, ISSN 2329-4949, doi:10.1145/3309206. Available from: https://doi.org/10.1145/3309206

[46] Scouarnec, N. L. Cuckoo++ Hash Tables: High-Performance Hash Tables for Networking Applications. *CoRR*, volume abs/1712.09624, 2017, 1712.09624. Available from: http://arxiv.org/abs/1712.09624

[47] CLHT. Accessed 4/1/2020. Available from: https://github.com/LPD-EPFL/CLHT

[48] Folly: Facebook Open-source Library. Accessed 29/12/2019. Available from: https://github.com/facebook/folly

[49] libcuckoo: A high-performance, concurrent hash table. Accessed 4/1/2020. Available from: https://github.com/efficient/libcuckoo

[50] Hopscotch Hashing - C++ Concurrency Package. Accessed 4/1/2020. Available from: https://sites.google.com/site/cconcurrencypackage/hopscotch-hashing

[51] junction: Concurrent data structures in C++. Accessed 4/1/2020. Available from: https://github.com/preshing/junction

[52] growt: a header only library offering a variety of dynamically growing concurrent hash tables. Accessed 4/1/2020. Available from: https://github.com/TooBiased/growt

[53] Kroilov, V. DeferredLRU benchmark application. Accessed 29/12/2019. Available from: https://github.com/metopa/lru_benchmark

[54] Kroilov, V. DeferredLRU - Highly scalable concurrent cache. 2019, accessed 29/12/2019. Available from: https://metopa.github.io/deferred%5Flru/

[55] Harris, T. L. A Pragmatic Implementation of Non-blocking Linked-Lists. In *Proceedings of the 15th International Conference on Distributed Computing*, DISC '01, London, UK, UK: Springer-Verlag, 2001, ISBN 3-540-42605-1, pp. 300–314. Available from: http://dl.acm.org/citation.cfm?id=645958.676105

[56] Dytrych, T. LSU3shell, Ab Initio No-Core Shell Model in SU(3)-scheme Basis. Accessed 29/12/2019. Available from: https://sourceforge.net/projects/lsu3shell

[57] Zipf, G. *Relative Frequency as a Determinant of Phonetic Change.* 1929, reprinted from the Harvard Studies in Classical Philology.

[58] Breslau, L.; Cao, P.; et al. Web Caching and Zipf-Like Distributions: Evidence and Implications. 4 1999, ISBN 0-7803-5417-6, pp. 126–134 vol.1, doi:10.1109/INFCOM.1999.749260.

[59] Hsu, W.; Smith, A. J. Characteristics of I/O Traffic in Personal Computer and Server Workloads. *IBM Syst. J.*, volume 42, no. 2, Apr. 2003: p. 347–372, ISSN 0018-8670, doi:10.1147/sj.422.0347. Available from: https://doi.org/10.1147/sj.422.0347

[60] Data traces from ARC: A Self-Tuning, Low Overhead Replacement Cache. Accessed 29/12/2019. Available from: https://researcher.watson.ibm.com/researcher/view%5Fperson%5Fsubpage.php?id=4700

[61] Hsu, W. W.; Smith, A. J.; et al. The Automatic Improvement of Locality in Storage Systems. *ACM Trans. Comput. Syst.*, volume 23, no. 4, Nov. 2005: p. 424–473, ISSN 0734-2071, doi:10.1145/1113574.1113577. Available from: https://doi.org/10.1145/1113574.1113577

[62] Urdaneta, G.; Pierre, G.; et al. Wikipedia Workload Analysis for Decentralized Hosting. *Elsevier Computer Networks*, volume 53, no. 11, July 2009: pp. 1830–1845, accessed 29/12/2019. Available from: http://www.globule.org/publi/WWADH%5Fcomnet2009.html

[63] Urdaneta, G.; Pierre, G.; et al. Wikipedia access traces. Accessed 29/12/2019. Available from: http://www.wikibench.eu/?page_id=60

[64] UMassTraceRepository. Accessed 29/12/2019. Available from: http://traces.cs.umass.edu

[65] Kelly, R.; Pearlmutter, B. A.; et al. Concurrent Robin Hood Hashing. In *OPODIS*, 2018.

[66] Abseil: Swiss Tables and `absl::Hash`. Accessed 29/12/2019. Available from: https://abseil.io/blog/20180927-swisstables

Appendix A

DeferredLRU Meta-parameter Measurements

A.1 Wikipedia 1/10

A.1.1 1 thread

Table A.1: DeferredLRU throughput with different pull threshold and purge threshold on Wikipedia1/10 trace for 1 thread

Pull threshold	Purge threshold					
	0.001	0.01	0.1	0.4	0.7	0.9
0.001	0.80	0.77	0.78	0.82	0.87	0.89
0.01	0.83	0.81	0.82	0.87	0.92	0.94
0.1	0.86	0.84	0.85	0.91	0.96	0.98
0.4	0.13	0.43	0.80	0.92	1.00	1.00
0.7	0.07	0.24	0.67	0.88	0.92	0.92
0.9	0.07	0.22	0.63	0.79	0.81	0.82

[1] Values are normalized by the maximum (3075915 op/s)

Table A.2: DeferredLRU hit-rate with different pull threshold and purge threshold on Wikipedia 1/10 trace for 1 thread

Pull	Purge threshold					
threshold	0.001	0.01	0.1	0.4	0.7	0.9
0.001	82.85%	82.93%	82.58%	81.25%	79.54%	78.11%
0.01	82.77%	82.80%	82.47%	81.11%	79.44%	78.05%
0.1	82.81%	82.82%	82.50%	81.23%	79.68%	78.55%
0.4	84.59%	83.33%	83.38%	82.19%	81.13%	80.85%
0.7	83.84%	85.31%	84.35%	83.24%	82.64%	82.69%
0.9	83.84%	85.98%	84.77%	84.01%	83.70%	83.60%

Table A.3: Baseline values for threshold and hit-rate on Wikipedia 1/10 trace for 1 thread

Cache	Throughput	Hit-rate
LRU	0.84	82.72%
HHVM	0.80	82.84%

[1] Throughput values are normalized by the maximum of Table A.1 (3075915 op/s)

A.1.2 32 threads

Table A.4: DeferredLRU throughput with different pull threshold and purge threshold on Wikipedia 1/10 trace for 32 threads

Pull threshold	Purge threshold					
	0.001	0.01	0.1	0.4	0.7	0.9
0.001	0.86	0.88	0.96	0.88	0.91	1.00
0.01	0.87	0.86	0.87	0.92	0.96	0.97
0.1	0.87	0.89	0.85	0.90	0.93	0.97
0.4	0.10	0.35	0.76	0.86	0.91	0.96
0.7	0.05	0.18	0.57	0.83	0.96	1.00
0.9	0.05	0.16	0.53	0.82	0.95	0.97

[1] Values are normalized by the maximum (135330 op/s)

Table A.5: DeferredLRU hit-rate with different pull threshold and purge threshold on Wikipedia 1/10 trace for 32 threads

Pull threshold	Purge threshold					
	0.001	0.01	0.1	0.4	0.7	0.9
0.001	82.84%	82.83%	82.71%	82.96%	83.17%	83.11%
0.01	82.84%	82.86%	82.84%	82.99%	83.06%	83.15%
0.1	82.96%	82.91%	82.95%	83.05%	83.14%	83.19%
0.4	84.59%	83.24%	83.49%	83.53%	83.56%	83.40%
0.7	83.86%	85.28%	84.39%	84.33%	84.16%	84.12%
0.9	83.86%	85.90%	84.87%	84.81%	84.65%	84.63%

Table A.6: Baseline values for threshold and hit-rate on Wikipedia 1/10 trace for 32 threads

Cache	Throughput	Hit-rate
LRU	0.19	83.77%
HHVM	6.14	82.18%

[1] Throughput values are normalized by the maximum of Table A.4 (135330 op/s)

A.2 Wikipedia 1/1000

A.2.1 1 thread

Table A.7: DeferredLRU throughput with different pull threshold and purge threshold on Wikipedia 1/1000 trace for 1 thread

Pull threshold	Purge threshold					
	0.001	0.01	0.1	0.4	0.7	0.9
0.001	0.83	0.83	0.84	0.86	0.88	0.88
0.01	0.86	0.88	0.89	0.90	0.92	0.93
0.1	0.89	0.90	0.92	0.94	0.96	0.98
0.4	0.22	0.50	0.86	0.95	0.99	1.00
0.7	0.14	0.36	0.81	0.94	0.96	0.96
0.9	0.11	0.30	0.76	0.86	0.87	0.88

[1] Values are normalized by the maximum (7889640 op/s)

Table A.8: DeferredLRU hit-rate with different pull threshold and purge threshold on Wikipedia 1/1000 trace for 1 thread

Pull	Purge threshold					
threshold	0.001	0.01	0.1	0.4	0.7	0.9
0.001	56.92%	56.89%	56.64%	55.71%	54.37%	52.85%
0.01	56.88%	56.85%	56.61%	55.68%	54.34%	52.87%
0.1	57.20%	57.17%	56.96%	56.21%	55.19%	53.93%
0.4	59.38%	59.28%	59.18%	58.87%	58.49%	58.37%
0.7	60.55%	60.50%	60.50%	60.29%	60.10%	60.07%
0.9	61.25%	61.17%	61.17%	61.00%	60.88%	60.86%

Table A.9: Baseline values for threshold and hit-rate on Wikipedia 1/1000 trace for 1 thread

Cache	Throughput	Hit-rate
LRU	1.22	56.87%
HHVM	0.60	56.91%

[1] Throughput values are normalized by the maximum of Table A.7 (7889640 op/s)

A.2.2 32 threads

Table A.10: DeferredLRU throughput with different pull threshold and purge threshold on Wikipedia 1/1000 trace for 32 threads

Pull threshold	Purge threshold					
	0.001	0.01	0.1	0.4	0.7	0.9
0.001	0.82	0.86	0.94	0.93	0.96	0.93
0.01	0.85	0.85	0.93	0.94	0.95	0.88
0.1	0.89	0.89	0.93	0.96	0.98	0.91
0.4	0.43	0.68	0.93	0.98	0.99	0.95
0.7	0.28	0.58	0.90	1.00	0.97	0.96
0.9	0.22	0.49	0.83	0.97	0.98	0.99

[1] Values are normalized by the maximum (62564 op/s)

Table A.11: DeferredLRU hit-rate with different pull threshold and purge threshold on Wikipedia 1/1000 trace for 32 threads

Pull	Purge threshold					
threshold	0.001	0.01	0.1	0.4	0.7	0.9
0.001	56.81%	56.86%	57.05%	57.32%	57.56%	57.66%
0.01	56.84%	56.85%	57.04%	57.32%	57.56%	57.61%
0.1	57.25%	57.25%	57.34%	57.50%	57.58%	57.61%
0.4	59.57%	59.23%	59.39%	59.43%	59.44%	59.44%
0.7	61.10%	60.62%	60.68%	60.73%	60.72%	60.70%
0.9	61.78%	61.34%	61.40%	61.47%	61.47%	61.47%

Table A.12: Baseline values for threshold and hit-rate on Wikipedia 1/1000 trace for 32 threads

Cache	Throughput	Hit-rate
LRU	0.47	57.10%
HHVM	10.69	56.36%

[1] Throughput values are normalized by the maximum of Table A.10 (62564 op/s)

A.3 P4 1/10

A.3.1 1 thread

Table A.13: DeferredLRU throughput with different pull threshold and purge threshold on P4 1/10 trace for 1 thread

Pull threshold	Purge threshold					
	0.001	0.01	0.1	0.4	0.7	0.9
0.001	0.97	0.93	0.90	0.94	0.98	1.00
0.01	0.93	0.90	0.88	0.93	0.96	0.99
0.1	0.56	0.82	0.86	0.91	0.95	0.98
0.4	0.13	0.42	0.81	0.92	0.98	0.98
0.7	0.07	0.23	0.66	0.88	0.90	0.91
0.9	0.05	0.19	0.63	0.79	0.83	0.82

[1] Values are normalized by the maximum (2099025 op/s)

Table A.14: DeferredLRU hit-rate with different pull threshold and purge threshold on P4 1/10 trace for 1 thread

Pull threshold	Purge threshold 0.001	0.01	0.1	0.4	0.7	0.9
0.001	48.53%	48.31%	47.40%	44.91%	41.33%	38.61%
0.01	48.40%	47.78%	47.55%	45.45%	41.43%	38.79%
0.1	48.02%	48.69%	47.74%	45.48%	41.34%	38.86%
0.4	47.44%	48.95%	48.79%	46.91%	43.10%	42.04%
0.7	45.48%	49.27%	49.69%	47.18%	45.16%	45.30%
0.9	44.21%	49.98%	49.75%	47.00%	46.93%	45.82%

Table A.15: Baseline values for threshold and hit-rate on P4 1/10 trace for 1 thread

Cache	Throughput	Hit-rate
LRU	1.46	48.29%
HHVM	0.73	48.17%

[1] Throughput values are normalized by the maximum of Table A.13 (2099025 op/s)

A.3.2 32 threads

Table A.16: DeferredLRU throughput with different pull threshold and purge threshold on P4 1/10 trace for 32 threads

Pull threshold	Purge threshold					
	0.001	0.01	0.1	0.4	0.7	0.9
0.001	0.99	0.99	0.99	1.00	0.95	0.90
0.01	0.94	0.98	0.98	0.99	0.95	0.91
0.1	0.59	0.90	0.97	0.98	0.94	0.96
0.4	0.06	0.39	0.85	0.93	0.92	0.94
0.7	0.02	0.17	0.68	0.88	0.92	0.95
0.9	0.03	0.15	0.56	0.84	0.87	0.91

[1] Values are normalized by the maximum (44123 op/s)

Table A.17: DeferredLRU hit-rate with different pull threshold and purge threshold on P4 1/10 trace for 32 threads

Pull	Purge threshold					
threshold	0.001	0.01	0.1	0.4	0.7	0.9
0.001	47.73%	47.54%	47.45%	45.50%	43.12%	42.16%
0.01	47.38%	47.05%	47.28%	45.70%	43.98%	42.23%
0.1	47.32%	47.67%	47.71%	45.23%	43.34%	42.86%
0.4	47.57%	48.02%	47.69%	46.16%	43.68%	43.59%
0.7	45.78%	48.48%	49.39%	47.06%	45.91%	45.95%
0.9	43.18%	49.68%	48.40%	46.62%	45.77%	46.34%

Table A.18: Baseline values for threshold and hit-rate on P4 1/10 trace for 32 threads

Cache	Throughput	Hit-rate
LRU	0.60	43.46%
HHVM	12.07	47.87%

[1] Throughput values are normalized by the maximum of Table A.16 (44123 op/s)

A.4 P4 1/1000

A.4.1 1 thread

Table A.19: DeferredLRU throughput with different pull threshold and purge threshold on P4 1/1000 trace for 1 thread

Pull threshold	Purge threshold					
	0.001	0.01	0.1	0.4	0.7	0.9
0.001	0.94	0.94	0.95	0.97	0.99	1.00
0.01	0.92	0.94	0.95	0.97	0.99	1.00
0.1	0.70	0.83	0.93	0.97	0.99	1.00
0.4	0.26	0.50	0.85	0.94	0.97	0.98
0.7	0.15	0.35	0.76	0.91	0.92	0.93
0.9	0.13	0.31	0.72	0.85	0.85	0.86

[1] Values are normalized by the maximum (6054175 op/s)

Table A.20: DeferredLRU hit-rate with different pull threshold and purge threshold on P4 1/1000 trace for 1 thread

Pull	Purge threshold					
threshold	0.001	0.01	0.1	0.4	0.7	0.9
0.001	3.41%	3.40%	3.37%	3.33%	3.21%	3.07%
0.01	3.41%	3.41%	3.37%	3.32%	3.20%	3.07%
0.1	3.42%	3.41%	3.39%	3.33%	3.19%	3.01%
0.4	3.49%	3.48%	3.42%	3.32%	3.10%	2.99%
0.7	3.54%	3.44%	3.34%	3.13%	2.99%	2.92%
0.9	3.22%	3.29%	3.16%	2.83%	2.75%	2.74%

Table A.21: Baseline values for threshold and hit-rate on P4 1/1000 trace for 1 thread

Cache	Throughput	Hit-rate
LRU	2.12	3.41%
HHVM	0.55	3.41%

[1] Throughput values are normalized by the maximum of Table A.19 (6054175 op/s)

A.4.2 32 threads

Table A.22: DeferredLRU throughput with different pull threshold and purge threshold on P4 1/1000 trace for 32 threads

Pull threshold	Purge threshold					
	0.001	0.01	0.1	0.4	0.7	0.9
0.001	0.88	0.94	0.95	0.90	0.95	0.95
0.01	0.90	0.95	0.93	1.00	0.94	0.96
0.1	0.84	0.87	0.96	0.92	0.98	0.97
0.4	0.51	0.72	0.89	0.93	0.92	0.93
0.7	0.34	0.57	0.88	0.94	0.93	0.92
0.9	0.24	0.47	0.82	0.92	0.91	0.91

[1] Values are normalized by the maximum (28026 op/s)

Table A.23: DeferredLRU hit-rate with different pull threshold and purge threshold on P4 1/1000 trace for 32 threads

Pull threshold	Purge threshold					
	0.001	0.01	0.1	0.4	0.7	0.9
0.001	0.74%	0.77%	0.73%	0.75%	0.74%	0.74%
0.01	0.77%	0.75%	0.74%	0.75%	0.73%	0.77%
0.1	0.79%	0.77%	0.77%	0.77%	0.78%	0.79%
0.4	0.86%	0.88%	0.86%	0.86%	0.86%	0.86%
0.7	1.05%	0.99%	0.97%	0.97%	0.97%	0.97%
0.9	1.07%	1.10%	1.08%	1.08%	1.07%	1.07%

Table A.24: Baseline values for threshold and hit-rate on P4 1/1000 trace for 32 threads

Cache	Throughput	Hit-rate
LRU	0.92	0.74%
HHVM	15.82	0.62%

[1] Throughput values are normalized by the maximum of Table A.22 (28026 op/s)

A.5 P8 1/10

A.5.1 1 thread

Table A.25: DeferredLRU throughput with different pull threshold and purge threshold on P8 1/10 trace for 1 thread

Pull threshold	Purge threshold					
	0.001	0.01	0.1	0.4	0.7	0.9
0.001	0.99	1.00	0.91	0.85	0.85	0.83
0.01	0.96	0.98	0.91	0.86	0.81	0.85
0.1	0.84	0.94	0.88	0.83	0.79	0.83
0.4	0.18	0.59	0.78	0.84	0.82	0.85
0.7	0.04	0.23	0.62	0.76	0.78	0.80
0.9	0.03	0.15	0.46	0.56	0.61	0.59

[1] Values are normalized by the maximum (3820560 op/s)

Table A.26: DeferredLRU hit-rate with different pull threshold and purge threshold on P8 1/10 trace for 1 thread

Pull	Purge threshold					
threshold	0.001	0.01	0.1	0.4	0.7	0.9
0.001	50.58%	50.45%	48.99%	43.20%	36.98%	32.62%
0.01	50.61%	50.48%	48.98%	43.19%	37.00%	32.62%
0.1	51.00%	50.86%	49.35%	43.87%	37.52%	33.22%
0.4	52.24%	51.88%	50.64%	45.44%	39.98%	38.78%
0.7	51.23%	53.31%	52.28%	47.65%	44.98%	44.26%
0.9	50.89%	53.66%	53.05%	50.64%	49.36%	48.77%

Table A.27: Baseline values for threshold and hit-rate on P8 1/10 trace for 1 thread

Cache	Throughput	Hit-rate
LRU	2.23	50.63%
HHVM	0.50	50.37%

[1] Throughput values are normalized by the maximum of Table A.25 (3820560 op/s)

A.5.2 32 threads

Table A.28: DeferredLRU throughput with different pull threshold and purge threshold on P8 1/10 trace for 32 threads

Pull threshold	Purge threshold					
	0.001	0.01	0.1	0.4	0.7	0.9
0.001	0.92	0.91	0.94	0.97	0.97	0.94
0.01	0.92	0.92	0.93	0.96	0.95	0.99
0.1	0.89	0.94	0.94	0.94	0.95	0.96
0.4	0.46	0.83	0.92	0.95	0.97	0.97
0.7	0.07	0.37	0.75	0.92	0.97	1.00
0.9	0.04	0.20	0.64	0.93	1.00	0.97

[1] Values are normalized by the maximum (51162 op/s)

Table A.29: DeferredLRU hit-rate with different pull threshold and purge threshold on P8 1/10 trace for 32 threads

Pull	Purge threshold					
threshold	0.001	0.01	0.1	0.4	0.7	0.9
0.001	49.99%	49.86%	49.82%	50.72%	51.04%	51.43%
0.01	49.50%	49.81%	50.16%	50.56%	50.99%	51.35%
0.1	50.21%	49.95%	49.87%	50.42%	51.12%	51.45%
0.4	52.08%	51.81%	51.71%	51.28%	51.54%	51.42%
0.7	50.80%	53.52%	53.13%	53.08%	52.97%	53.08%
0.9	54.69%	55.19%	55.15%	54.72%	54.72%	54.56%

Table A.30: Baseline values for threshold and hit-rate on P8 1/10 trace for 32 threads

Cache	Throughput	Hit-rate
LRU	0.56	49.17%
HHVM	11.81	45.98%

[1] Throughput values are normalized by the maximum of Table A.28 (51162 op/s)

A.6 P8 1/238

A.6.1 1 thread

Table A.31: DeferredLRU throughput with different pull threshold and purge threshold on P8 1/238 trace for 1 thread

Pull threshold	Purge threshold					
	0.001	0.01	0.1	0.4	0.7	0.9
0.001	0.93	0.94	0.96	0.98	0.99	1.00
0.01	0.93	0.94	0.96	0.98	0.99	1.00
0.1	0.89	0.91	0.94	0.97	0.99	0.99
0.4	0.68	0.74	0.89	0.95	0.97	0.98
0.7	0.49	0.56	0.81	0.92	0.94	0.94
0.9	0.40	0.47	0.77	0.84	0.85	0.85

[1] Values are normalized by the maximum (6287865 op/s)

Table A.32: DeferredLRU hit-rate with different pull threshold and purge threshold on P8 1/238 trace for 1 thread

Pull	Purge threshold					
threshold	0.001	0.01	0.1	0.4	0.7	0.9
0.001	0.34%	0.33%	0.33%	0.31%	0.28%	0.25%
0.01	0.30%	0.30%	0.30%	0.29%	0.27%	0.24%
0.1	0.32%	0.32%	0.31%	0.30%	0.27%	0.25%
0.4	0.37%	0.36%	0.36%	0.34%	0.31%	0.30%
0.7	0.40%	0.40%	0.39%	0.36%	0.35%	0.34%
0.9	0.41%	0.43%	0.42%	0.40%	0.38%	0.37%

Table A.33: Baseline values for threshold and hit-rate on P8 1/238 trace for 1 thread

Cache	Throughput	Hit-rate
LRU	2.22	0.34%
HHVM	0.53	0.32%

[1] Throughput values are normalized by the maximum of Table A.31 (6287865 op/s)

A.6.2 32 threads

Table A.34: DeferredLRU throughput with different pull threshold and purge threshold on P8 1/238 trace for 32 threads

Pull threshold	Purge threshold					
	0.001	0.01	0.1	0.4	0.7	0.9
0.001	0.93	0.93	0.96	0.98	0.98	0.96
0.01	0.93	0.95	0.96	0.97	0.95	0.99
0.1	0.89	0.91	0.95	1.00	0.98	0.98
0.4	0.74	0.80	0.91	0.97	0.95	0.98
0.7	0.66	0.74	0.90	0.96	0.96	0.99
0.9	0.63	0.70	0.88	0.92	0.97	0.97

[1] Values are normalized by the maximum (27324 op/s)

Table A.35: DeferredLRU hit-rate with different pull threshold and purge threshold on P8 1/238 trace for 32 threads

Pull	Purge threshold					
threshold	0.001	0.01	0.1	0.4	0.7	0.9
0.001	0.55%	0.56%	0.56%	0.57%	0.57%	0.56%
0.01	0.55%	0.55%	0.56%	0.58%	0.57%	0.57%
0.1	0.57%	0.58%	0.58%	0.58%	0.59%	0.59%
0.4	0.66%	0.67%	0.66%	0.67%	0.67%	0.66%
0.7	0.74%	0.75%	0.77%	0.76%	0.76%	0.76%
0.9	0.83%	0.85%	0.87%	0.87%	0.86%	0.85%

Table A.36: Baseline values for threshold and hit-rate on P8 1/238 trace for 32 threads

Cache	Throughput	Hit-rate
LRU	0.84	0.65%
HHVM	16.65	0.69%

[1] Throughput values are normalized by the maximum of Table A.34 (27324 op/s)

Appendix B

DeferredLRU Performance Measurements

Table B.1: Singular cache performance I

		Hit-rate		**Throughput**			
		Threads		Threads			
Trace	**Cache**	1	32	1	16	32	**Speedup**
DS1 1/10	DLRU 0.001|0.1	3.25%	4.90%	1.95	0.89	**0.85**	**0.44**
	DLRU 0.1|0.7	3.32%	5.52%	2.32	0.88	0.84	0.36
	DLRU 0.99|0.99	**6.73%**	**12.00%**	**3.62**	**0.90**	0.84	0.23
	LRU	3.43%	5.58%	3.07	0.73	0.67	0.22
	conLRU	3.55%	3.64%	1.56	0.66	0.50	0.32
	TBB	3.44%	3.51%	2.09	0.48	0.30	0.14
DS1 1/1000	DLRU 0.001|0.1	0.96%	0.21%	5.32	0.86	0.86	0.16
	DLRU 0.1|0.7	0.83%	0.22%	5.65	**0.87**	**0.87**	0.15
	DLRU 0.99|0.99	0.53%	**0.40%**	1.81	0.84	0.82	**0.45**
	LRU	0.97%	0.15%	**12.41**	0.83	0.76	0.06
	conLRU	**0.98%**	0.30%	4.32	0.66	0.50	0.12
	TBB	0.96%	0.34%	4.20	0.62	0.36	0.09
OLTP 1/10	DLRU 0.001|0.1	66.23%	**66.20%**	6.80	**2.54**	**2.24**	0.33
	DLRU 0.1|0.7	62.74%	57.55%	7.23	1.99	1.87	0.26
	DLRU 0.99|0.99	62.72%	59.26%	4.76	2.07	1.92	**0.40**
	LRU	66.70%	52.99%	**9.60**	0.98	0.89	0.09
	conLRU	**66.70%**	55.03%	5.14	0.91	0.62	0.12
	TBB	66.69%	64.62%	2.37	0.78	0.48	0.20
OLTP 1/45	DLRU 0.001|0.1	50.63%	32.51%	6.59	1.28	1.20	0.18
	DLRU 0.1|0.7	46.00%	33.33%	7.26	1.31	1.24	0.17
	DLRU 0.99|0.99	46.44%	41.99%	5.44	**1.41**	**1.40**	**0.26**
	LRU	51.27%	32.58%	**10.66**	0.94	0.82	0.08
	conLRU	**51.27%**	34.63%	5.42	0.75	0.56	0.10
	TBB	51.27%	**43.45%**	2.93	0.84	0.52	0.18
P4 1/10	DLRU 0.001|0.1	47.55%	**47.43%**	1.84	**1.53**	**1.43**	0.78
	DLRU 0.1|0.7	41.66%	42.73%	1.95	1.39	1.32	0.68
	DLRU 0.99|0.99	45.75%	44.83%	0.80	1.35	1.29	**1.60**
	LRU	**48.38%**	42.84%	**2.97**	0.89	0.79	0.27
	conLRU	48.34%	46.68%	1.68	0.86	0.61	0.36
	TBB	48.16%	45.94%	1.85	0.51	0.32	0.17
P4 1/1000	DLRU 0.001|0.1	3.38%	0.74%	5.73	0.85	0.84	0.15
	DLRU 0.1|0.7	3.19%	0.76%	5.97	0.88	**0.86**	0.14
	DLRU 0.99|0.99	1.92%	**1.13%**	3.00	0.86	0.83	**0.28**
	LRU	3.41%	0.70%	**12.84**	**0.90**	0.73	0.06
	conLRU	3.39%	0.83%	4.48	0.66	0.50	0.11
	TBB	**3.42%**	0.83%	3.55	0.66	0.40	0.11

Table B.2: Singular cache performance II

		Hit-rate		**Throughput**			
		Threads		Threads			
Trace	**Cache**	1	32	1	16	32	**Speedup**
P8 1/10	DLRU 0.001|0.1	48.99%	49.84%	3.54	1.69	1.54	0.44
	DLRU 0.1|0.7	37.52%	50.93%	3.13	1.68	1.56	0.50
	DLRU 0.99|0.99	**51.06%**	**56.10%**	1.04	**1.78**	**1.67**	**1.61**
	LRU	50.62%	48.61%	**8.45**	1.08	0.93	0.11
	conLRU	50.59%	51.01%	3.11	0.87	0.63	0.20
	TBB	50.43%	50.34%	1.95	0.61	0.38	0.20
P8 1/238	DLRU 0.001|0.1	0.69%	2.50%	5.73	0.88	0.86	0.15
	DLRU 0.1|0.7	0.65%	2.60%	5.94	0.91	**0.88**	0.15
	DLRU 0.99|0.99	**1.95%**	**3.87%**	2.63	**0.91**	0.87	**0.33**
	LRU	0.73%	2.57%	**12.45**	0.82	0.77	0.06
	conLRU	0.73%	2.50%	4.42	0.64	0.51	0.11
	TBB	0.73%	2.51%	3.32	0.69	0.41	0.12
S3 1/10	DLRU 0.001|0.1	3.77%	11.11%	1.56	0.92	**0.90**	0.58
	DLRU 0.1|0.7	3.09%	12.14%	1.54	**0.94**	0.89	0.58
	DLRU 0.99|0.99	**17.03%**	**16.85%**	1.17	0.86	0.85	**0.73**
	LRU	3.98%	10.60%	**3.72**	0.77	0.66	0.18
	conLRU	3.98%	4.31%	1.42	0.66	0.51	0.36
	TBB	3.97%	4.03%	1.31	0.41	0.25	0.19
S3 1/412	DLRU 0.001|0.1	0.08%	0.21%	2.63	0.87	0.89	0.34
	DLRU 0.1|0.7	0.06%	0.22%	3.28	**0.88**	**0.90**	0.28
	DLRU 0.99|0.99	**0.53%**	**0.44%**	2.37	0.87	0.88	**0.37**
	LRU	0.09%	0.25%	**3.74**	0.58	0.57	0.15
	conLRU	0.09%	0.10%	1.83	0.68	0.50	0.27
	TBB	0.09%	0.09%	2.51	0.69	0.44	0.17
Wikipedia 1/10	DLRU 0.001|0.1	82.72%	82.64%	2.24	5.24	3.89	1.74
	DLRU 0.1|0.7	79.72%	83.09%	**2.77**	**5.82**	4.22	1.52
	DLRU 0.99|0.99	**83.92%**	**85.10%**	1.43	5.13	**4.30**	**3.01**
	LRU	82.92%	82.91%	2.40	0.90	0.78	0.33
	conLRU	82.47%	83.03%	1.52	0.98	0.75	0.49
	TBB	83.81%	84.12%	0.83	0.43	0.29	0.35
Wikipedia 1/1000	DLRU 0.001|0.1	56.64%	57.09%	6.66	1.95	1.82	0.27
	DLRU 0.1|0.7	55.19%	57.58%	7.61	2.01	1.92	0.25
	DLRU 0.99|0.99	**61.30%**	**61.92%**	4.44	**2.19**	**2.09**	**0.47**
	LRU	56.88%	56.92%	**9.57**	1.00	0.82	0.09
	conLRU	56.93%	56.97%	5.18	0.86	0.67	0.13
	TBB	56.99%	57.35%	2.40	0.89	0.54	0.22

Table B.3: Singular cache performance III

		Hit-rate		Throughput			
		Threads		Threads			
Trace	**Cache**	1	32	1	16	32	**Speedup**
YouTube 1/10	DLRU 0.001|0.1	46.89%	**46.71%**	4.10	**1.55**	**1.50**	0.37
	DLRU 0.1|0.7	43.37%	38.76%	4.28	1.29	1.27	0.30
	DLRU 0.99|0.99	44.82%	39.61%	2.75	1.28	1.24	**0.45**
	LRU	**47.26%**	31.18%	**6.37**	0.85	0.77	0.12
	conLRU	47.26%	35.69%	3.43	0.82	0.57	0.16
	TBB	47.24%	46.42%	1.18	0.54	0.33	0.28
YouTube 1/120	DLRU 0.001|0.1	28.99%	7.79%	5.72	0.91	0.91	0.16
	DLRU 0.1|0.7	27.44%	8.08%	6.15	0.92	0.91	0.15
	DLRU 0.99|0.99	27.53%	13.90%	4.96	**0.96**	**0.96**	0.19
	LRU	29.25%	7.12%	**10.35**	0.78	0.72	0.07
	conLRU	**29.25%**	9.99%	4.75	0.66	0.51	0.11
	TBB	29.25%	**19.17%**	2.19	0.76	0.46	**0.21**
Zipf 1/10	DLRU 0.001|0.1	58.55%	59.40%	7.39	2.01	1.97	0.27
	DLRU 0.1|0.7	54.14%	60.28%	7.79	2.13	1.98	0.25
	DLRU 0.99|0.99	**66.86%**	**65.03%**	1.83	**2.41**	**2.21**	**1.20**
	LRU	59.26%	59.22%	**9.88**	0.97	0.88	0.09
	conLRU	59.26%	59.20%	5.35	0.89	0.69	0.13
	TBB	59.26%	59.25%	2.39	0.87	0.53	0.22
Zipf 1/22	DLRU 0.001|0.1	48.44%	49.21%	6.67	1.62	1.56	0.23
	DLRU 0.1|0.7	45.00%	50.10%	7.39	1.69	1.59	0.22
	DLRU 0.99|0.99	**55.83%**	**56.29%**	4.35	**1.88**	**1.80**	**0.41**
	LRU	49.08%	49.05%	**10.31**	0.92	0.89	0.09
	conLRU	49.08%	49.00%	5.31	0.82	0.65	0.12
	TBB	49.08%	49.10%	2.55	0.87	0.54	0.21

Table B.4: Binned cache performance I

		Hit-rate		**Throughput**			
		Threads		Threads			
Trace	**Cache**	1	32	1	16	32	**Speedup**
DS1 1/10	B-DLRU 0.001|0.1	**3.95%**	**8.36%**	1.00	12.45	21.83	21.75
	B-DLRU 0.1|0.7	3.95%	8.31%	1.00	12.54	22.05	22.07
	B-DLRU 0.99|0.99	3.93%	8.10%	1.00	12.48	21.86	21.79
	B-LRU	3.63%	6.87%	**1.53**	**19.09**	**26.32**	17.20
	B-conLRU	3.33%	5.91%	0.69	10.97	16.11	**23.23**
	HHVM	3.41%	5.76%	1.10	8.91	10.53	9.55
DS1 1/1000	B-DLRU 0.001|0.1	0.94%	**0.52%**	4.69	23.52	33.68	**7.18**
	B-DLRU 0.1|0.7	0.94%	0.43%	4.70	23.87	33.20	7.07
	B-DLRU 0.99|0.99	0.95%	0.51%	4.68	23.88	33.57	7.18
	B-LRU	0.97%	0.13%	**9.01**	**29.50**	**35.60**	3.95
	B-conLRU	0.98%	0.11%	3.88	14.80	18.04	4.65
	HHVM	**0.99%**	0.12%	2.96	13.63	13.62	4.61
OLTP 1/10	B-DLRU 0.001|0.1	66.05%	62.49%	6.30	**35.91**	52.54	**8.34**
	B-DLRU 0.1|0.7	66.05%	62.37%	6.32	35.50	52.08	8.25
	B-DLRU 0.99|0.99	66.05%	**62.63%**	6.32	35.35	**52.55**	8.32
	B-LRU	66.67%	55.38%	**8.73**	32.24	40.66	4.66
	B-conLRU	66.67%	59.10%	4.91	21.54	27.54	5.61
	HHVM	**66.69%**	62.08%	4.57	20.10	22.94	5.02
OLTP 1/45	B-DLRU 0.001|0.1	48.49%	32.27%	6.16	26.82	**37.10**	**6.02**
	B-DLRU 0.1|0.7	48.49%	32.36%	6.17	26.83	36.92	5.98
	B-DLRU 0.99|0.99	48.49%	32.35%	6.17	26.71	36.69	5.95
	B-LRU	51.13%	**34.79%**	**9.42**	**27.11**	34.45	3.66
	B-conLRU	51.14%	33.32%	5.12	16.47	20.30	3.96
	HHVM	**51.22%**	32.78%	4.75	16.11	17.20	3.62
P4 1/10	B-DLRU 0.001|0.1	47.98%	46.67%	1.37	23.38	39.35	28.77
	B-DLRU 0.1|0.7	47.98%	46.79%	1.37	23.67	**39.35**	**28.82**
	B-DLRU 0.99|0.99	**48.21%**	46.63%	1.38	23.68	39.27	28.52
	B-LRU	48.04%	36.65%	**2.14**	**26.66**	32.54	15.17
	B-conLRU	48.20%	46.26%	1.07	17.34	22.67	21.09
	HHVM	47.88%	**47.90%**	1.47	12.72	15.61	10.58
P4 1/1000	B-DLRU 0.001|0.1	3.34%	0.53%	4.95	23.76	32.15	6.50
	B-DLRU 0.1|0.7	3.34%	0.52%	4.96	23.76	32.41	**6.54**
	B-DLRU 0.99|0.99	3.34%	0.53%	4.96	23.71	32.42	6.54
	B-LRU	**3.42%**	0.56%	**9.60**	**28.24**	**33.03**	3.44
	B-conLRU	3.40%	0.57%	4.08	14.22	17.30	4.24
	HHVM	3.42%	**0.62%**	3.33	14.16	13.93	4.19

Table B.5: Binned cache performance II

		Hit-rate		**Throughput**			
		Threads		Threads			
Trace	**Cache**	1	32	1	16	32	**Speedup**
P8 1/10	B-DLRU 0.001|0.1	49.19%	44.41%	3.19	31.30	43.96	13.79
	B-DLRU 0.1|0.7	49.20%	44.43%	3.18	30.98	**44.06**	13.85
	B-DLRU 0.99|0.99	49.19%	44.40%	3.17	30.96	44.02	**13.88**
	B-LRU	**50.58%**	46.19%	**6.08**	**36.04**	41.07	6.75
	B-conLRU	50.49%	**46.28%**	2.62	22.13	26.77	10.22
	HHVM	50.40%	46.02%	1.94	16.97	18.60	9.58
P8 1/238	B-DLRU 0.001|0.1	0.72%	2.13%	4.94	23.73	32.00	6.48
	B-DLRU 0.1|0.7	0.72%	2.13%	4.94	23.73	32.20	**6.52**
	B-DLRU 0.99|0.99	0.72%	2.13%	4.94	23.74	31.35	6.34
	B-LRU	0.74%	2.43%	**9.58**	**27.78**	**33.38**	3.48
	B-conLRU	**0.74%**	2.41%	4.06	14.32	17.40	4.29
	HHVM	0.72%	**2.61%**	3.28	14.29	14.07	4.29
S3 1/10	B-DLRU 0.001|0.1	**4.16%**	13.32%	0.63	15.02	23.56	37.33
	B-DLRU 0.1|0.7	4.16%	13.37%	0.63	14.95	23.72	**37.71**
	B-DLRU 0.99|0.99	4.15%	13.47%	0.63	14.91	**23.72**	37.62
	B-LRU	3.97%	**13.79%**	**1.05**	**15.39**	20.75	19.83
	B-conLRU	3.95%	13.72%	0.44	10.56	15.13	34.07
	HHVM	3.98%	13.51%	0.59	8.99	11.17	18.96
S3 1/412	B-DLRU 0.001|0.1	0.07%	0.27%	3.01	12.88	**18.21**	**6.04**
	B-DLRU 0.1|0.7	0.07%	0.27%	3.01	13.08	16.86	5.61
	B-DLRU 0.99|0.99	0.07%	0.27%	3.01	13.01	16.75	5.57
	B-LRU	0.09%	**0.32%**	**3.45**	**13.53**	17.98	5.21
	B-conLRU	0.09%	0.28%	2.07	7.79	6.85	3.32
	HHVM	**0.09%**	0.31%	2.67	11.26	12.25	4.59
Wikipedia 1/10	B-DLRU 0.001|0.1	82.71%	81.81%	2.38	35.71	55.12	23.12
	B-DLRU 0.1|0.7	82.73%	81.80%	2.36	35.35	**57.04**	**24.18**
	B-DLRU 0.99|0.99	82.71%	81.78%	2.38	**35.72**	56.98	23.98
	B-LRU	**82.99%**	82.21%	2.33	24.02	31.75	13.63
	B-conLRU	82.44%	**82.39%**	1.46	18.19	24.45	16.69
	HHVM	82.92%	82.14%	**2.40**	20.06	25.53	10.65
Wikipedia 1/1000	B-DLRU 0.001|0.1	56.59%	56.02%	6.72	35.28	**51.39**	**7.65**
	B-DLRU 0.1|0.7	56.59%	55.99%	6.72	**35.42**	50.43	7.51
	B-DLRU 0.99|0.99	56.59%	56.00%	6.72	35.04	51.29	7.63
	B-LRU	56.84%	56.34%	**8.92**	27.94	36.76	4.12
	B-conLRU	**56.93%**	**56.53%**	5.10	18.53	23.17	4.55
	HHVM	56.87%	56.37%	4.56	18.67	20.78	4.55

Table B.6: Binned cache performance III

		Hit-rate		**Throughput**			
		Threads		Threads			
Trace	**Cache**	1	32	1	16	32	**Speedup**
YouTube 1/10	B-DLRU 0.001|0.1	46.93%	39.84%	4.13	27.28	40.29	9.75
	B-DLRU 0.1|0.7	46.93%	41.08%	4.18	27.54	**40.97**	**9.80**
	B-DLRU 0.99|0.99	46.93%	39.14%	4.18	27.59	39.44	9.44
	B-LRU	**47.24%**	33.11%	**6.13**	**29.01**	39.35	6.42
	B-conLRU	47.24%	35.41%	3.41	19.23	24.03	7.05
	HHVM	47.24%	**43.71%**	2.32	16.04	17.75	7.64
YouTube 1/120	B-DLRU 0.001|0.1	28.39%	7.42%	5.48	22.99	31.93	**5.83**
	B-DLRU 0.1|0.7	28.39%	7.20%	5.48	23.22	31.52	5.75
	B-DLRU 0.99|0.99	28.39%	7.17%	5.48	23.35	31.29	5.71
	B-LRU	29.22%	**8.65%**	**9.51**	**26.14**	**32.50**	3.42
	B-conLRU	29.22%	7.54%	4.60	14.54	17.41	3.79
	HHVM	**29.24%**	7.36%	3.89	13.07	13.64	3.51
Zipf 1/10	B-DLRU 0.001|0.1	57.86%	57.95%	6.94	**36.59**	51.12	**7.37**
	B-DLRU 0.1|0.7	57.86%	57.94%	6.97	36.11	50.75	7.28
	B-DLRU 0.99|0.99	57.86%	57.95%	7.00	35.81	**51.24**	7.33
	B-LRU	59.22%	**59.18%**	**8.98**	25.38	18.20	2.03
	B-conLRU	59.22%	59.17%	5.16	9.22	4.24	0.82
	HHVM	**59.26%**	58.90%	4.74	18.84	21.72	4.58
Zipf 1/22	B-DLRU 0.001|0.1	46.86%	47.01%	6.45	**31.11**	**42.62**	**6.61**
	B-DLRU 0.1|0.7	46.86%	47.01%	6.46	30.81	42.24	6.53
	B-DLRU 0.99|0.99	46.86%	47.01%	6.47	31.00	41.88	6.47
	B-LRU	49.01%	**48.97%**	**9.27**	21.64	16.51	1.78
	B-conLRU	49.01%	48.95%	5.05	8.60	4.08	0.81
	HHVM	**49.08%**	48.75%	4.66	17.72	19.37	4.16

Appendix C

Contents of SD card

`thesis_kroilov_viacheslav_2020`	Root directory
└ `text`	Thesis text directory
└ `chapter`	Chapter LaTeX directory
└ `appendix*.tex`	Appendix chapters from the thesis
└ `figures`	Figures directory
└ `*.tex`	LaTeX files for figures used in the thesis
└ `listings`	Listings directory
└ `*.tex`	LaTeX files for code listings used in the thesis
└ `plots`	Plots directory
└ `*`	LaTeX files for plots used in the thesis
└ `tables`	Tables directory
└ `*.tex`	LaTeX files for tables used in the thesis
└ `FITthesis.cls`	LaTeX style file for the thesis
└ `index.bib`	Entities cited in the thesis in BibTex format
└ `Thesis_Kroilov_Viacheslav_2020.pdf`	the thesis in PDF format
└ `Thesis_Kroilov_Viacheslav_2020.tex`	the thesis in LaTeX format
└ `traces`	Data traces directory
└ `DS1.blis`	DS1 trace
└ `OLTP.blis`	OLTP trace
└ `P4.blis`	P4 trace
└ `P8.blis`	P8 trace
└ `S3.blis`	S3 trace
└ `trace_info.py`	Python utility for gathering trace stats
└ `wiki_dataset.blis`	Wikipedia trace
└ `youtube.blis`	YouTube trace
└ `zipf_09.blis`	Zipf trace