CZECH TECHNICAL UNIVERSITY IN PRAGUE
FACULTY OF INFORMATION TECHNOLOGY

ASSIGNMENT OF BACHELOR'S THESIS

Title:	Numerical database system
Student:	Viacheslav Kroilov
Supervisor:	doc. Ing. Ivan Šimeček, Ph.D.
Study Programme:	Informatics
Study Branch:	Computer Science
Department:	Department of Theoretical Computer Science
Validity:	Until the end of summer semester 2017/18

Instructions

1) Study and explore the current concept of numerical database system (see [1] and [2]) that stores most frequent search terms in a weighted search tree.

2) Discuss advantages and drawbacks of different types of weighted search trees in application to the algorithm.

3) Explore different strategies for storing terms.

4) Implement a parallel version of the described algorithm.

5) Perform a performance measurement (throughput and latency) of the parallel version on a multicore system. Tests are based on common usage scenarios of this algorithm.

6) Implement a ready-to-use open source library in C++ programming language.

References

[1] S. C. Parkb, C. Bahria, J. P. Draayerb, S. -Q. Zhengb: Numerical database system based on a weighted search tree, Computer Physics Communications, Volume 82, Issues 2-3, September 1994, Pages 247-264.

[2] CTU FIT Bachelor Thesis 2016, Miroslav Mašat: Numerical database system, Prague, February 4.

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

prof. Ing. Pavel Tvrdík, CSc.
Dean

Prague February 11, 2017

Czech Technical University in Prague
Faculty of Information Technology
Department of Theoretical Computer Science

Bachelor's thesis

Concurrent Memoization System for Large-Scale Scientific Applications

Viacheslav Kroilov

Supervisor: doc. Ing. Ivan Šimeček, Ph.D.

16th May 2017

Acknowledgements

First of all, I would like to thank my parents, Mikhail Kroilov and Elena Kroilova, for the constant support they are giving to me.

Also I would like to thank Valerie Bashkirtseva for the warmth she gives to me and for making the place we are living in the home.

Finally, but not least, I want to thank my thesis supervisor doc. Ing. Ivan Šimeček, Ph.D. for the valuable feedbacks on preliminary versions of this thesis and for the experience I have gained during the BI-EIA course.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as school work under the provisions of Article 60(1) of the Act.

In Prague on 16th May 2017 .....

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Kroilov, Viacheslav. *Concurrent Memoization System for Large-Scale Scientific Applications.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2017.

Abstrakt

Numerická databáze zrychluje výpočet ukládáním mezivýsledků do pamětí. Kanonická implementace numerické databáze je založená na ohodnoceném binárním stromu – kombinace AVL-stromu a binární haldy. V této práci je diskutována i možnost využití jiných datových struktur, jak je Splay-strom a hašovácí tabulka. Navíc je zavedená zcela nová datová struktura – CNDC. Podporuje stejné operace jako ohodnocený binární strom, ale je přizpůsobená k použití ve vícevláknovém prostředí.

Všechny zmíněné datové struktury jsou implementovány v programovacím jazyce C++ v podobě programovací knihovny NUMDB. Na závěr jsou uvedené výsledky měření výkonnosti implementovaných datových struktur.

Klíčová slova numerická databáze, vypočetní optimizace, splay strom, hašovací tabulka, datové struktury pro paralelní zpracování, vícevláknová synchronizace, fine-grained locking, C++

Abstract

Numerical databases speed up computations by memoizing pairs of an argument and the result, computed by a function with the argument. The canonical numerical database is based on the weighted search tree – a combination of the AVL tree and the binary heap. The application of alternative data structures, namely the hash table and the splay tree, is discussed in this thesis. In addition, a new data structure – CNDC – is introduced. It is similar to the weighted search tree, but all operations are declared as thread-safe.

Data structures, mentioned above, are implemented in the C++ programming language as a programming library, called NUMDB. The performance of each data structure is measured, and the results are compared and discussed.

Keywords numerical database, computational optimization, splay tree, hash table, concurrent lookup data structure, thread synchronisation, fine-grained locking, C++

Contents

List of Tables

List of Figures

List of Algorithms

List of Listings

Introduction

There is a strong trend nowadays in scientific computations – the size of data to be processed increases faster than available computing resources. Different optimization techniques are used to overcome this problem. For example, a memoization technique, which is as follows – a result R that has been computed once by a function F with the argument A is stored in memory (together with A). Then on consequent calls of F with A, R is retrieved from memory, hence the actual call to F is omitted. This optimization turns out to be particularly useful when a result of F takes a long time to be computed (relatively to a time needed to retrieve R from memory), and F is usually called only with a small subset of arguments. Sometimes this technique is also referred to as a caching.

A target audience for this kind of memoization systems is scientific application developers.

The concept of a numerical database, introduced by Park, Draayer, and Zheng[1], describes one of such memoization systems. Its main purpose is to "store and retrieve valuable intermediate information so costly redundant calculations can be avoided." Authors also presented a possible implementation, that is based on a weighted search tree.

The main goal of this thesis is to explore different look-up data structures, especially those supporting a concurrent access, which can be used in place of the weighted search tree and compare their performance. Then, the most promising data structures will be packed in a programming library, written in C++ programming language.

Chapter 1

Preliminaries

Essential programming concepts and data structures are described in this chapter. The rest of the thesis relies on these terms.

1.1 Related Data Structures

1.1.1 Binary Search Tree

A binary search tree is a data structure that implements ***find***(), ***insert***(), and ***remove***() operations on a set of keys. Key K can be of any type, that has a total order. Throughout the thesis, trees with distinct keys are discussed, although of course, it is not the necessary condition.

A binary tree (not to be confused with a binary search tree) consists of nodes. A node N consists of a key (defined as $\boldsymbol{key}(N)$) and two references to other nodes – left child (defined as $\boldsymbol{left}(N)$) and right child (defined as $\boldsymbol{right}(N)$). A reference may contain a link to an existing node or a special value nil, that means that the reference is empty. A node that contains a link to a child is called a *parent* of this child node. Nodes may contain other attributes as well, but those are not substantial for this explanation. A node that has no children ($\boldsymbol{left}(N) = nil \wedge \boldsymbol{right}(N) = nil$) is called a *leaf*.

From the perspective of the graph theory, the binary tree is a simple oriented acyclic graph, where vertices are represented as nodes and edges are represented as links between a node and its left and right children. Every vertex in such graph has at most one incoming edge, i.e. every node can have at most one parent. Moreover, only one node has no parent – this node is called the *root* of a binary tree. The length of the longest path from any leaf to the root is known as the *height* of a binary tree. The *subtree* with a root in N defined as N and a set of nodes that can be reached from N by child links.

Algorithm 1 Lookup in BST

```
1: procedure FIND(root, K)                    ▷ The node with key = K or nil
2:     node ← root
3:     while node ≠ nil do
4:         if K = key(node) then
5:             return node
6:         else if K < key(node) then
7:             node ← left(node)
8:         else                                ▷ K > key(node)
9:             node ← right(node)
10:        end if
11:    end while
12:    return nil                             ▷ The node was not found
13: end procedure
```

A binary *search* tree (BST) is a binary tree that satisfies the following condition: for each N, subtrees with the root in $\textbf{\textit{left}}(N)$ and with the root in $\textbf{\textit{right}}(N)$ contain only nodes with keys, that are less or equal than $\textbf{\textit{key}}(N)$ and larger or equal than $\textbf{\textit{key}}(N)$, respectively. Using this property, it is possible to implement a fast lookup of a key K in a binary search tree (see Algorithm 1).

The complexity of this search algorithm is $\mathcal{O}(\textbf{\textit{height}}(T))$ (assuming that key comparison takes $\mathcal{O}(1)$). Furthermore, two remaining operations of a binary search tree, ***insert***() and ***remove***(), are implemented in the same fashion, and both of those operations have complexity $\mathcal{O}(\textbf{\textit{height}}(T))$ as well. Their implementation is described in detail in [2, p. 327].

The tree height can vary between $\mathcal{O}(\textbf{\textit{size}}(T))$, and $\mathcal{O}(\log(\textbf{\textit{size}}(T)))$ in case of a complete binary tree [3], where $\textbf{\textit{size}}(T)$ is the count of nodes in T. BST will maintain optimal operation time only if its structure is close to a complete binary tree and the height is bounded by $c \times \log(\textbf{\textit{size}}(T))$, where c is a constant factor greater or equal 1.

To keep the height logarithmic, even in a worst-case scenario, the *tree rebalancing* has been invented. The idea is that a tree keeps track of its structure and if it is not optimal, then the rebalancing is applied to restore optimal structure. The rebalancing can be achieved with the *tree rotation*[2, p. 435] – the operation, that swaps a node with its parent in a way, that preserves the BST property.

1.1.2 AVL Tree

AVL tree was invented in 1962 by Georgy Adelson-Velsky and Evgenii Landis[4]. It is a classic example of a self-balancing BST. In fact, the height of the AVL tree is never greater than $1.4405 \times \log(\textbf{\textit{size}}(T)) - 0.3277$ [5, p. 460].

Self-balancing is achieved with the following approach: every node holds the difference between the heights of its left and right subtrees; this difference is called *balance factor*. *AVL property* requires the balance factor of every node to be in range of values -1, 0 and 1.

After every operation, that modifies the tree structure – ***insert***() and ***remove***(), balance factors are updated. If at any step the balance factor happens to be -2 or 2, a rotation or a double rotation is applied. The rotation adjusts the heights of the left and right subtrees and, consequently, restores the AVL property. The exact AVL tree implementation is described in [5, p. 458].

1.1.3 Splay Tree

Another approach on tree balancing is presented in the Sleator and Tarjan work[6] – "The efficiency of splay trees comes not from an explicit structural constraint, as with balanced trees, but from applying a simple restructuring heuristic, called splaying, whenever the tree is accessed." Term *splaying* stands for the process of using rotations (similar to ones in the AVL tree) to bring the last accessed node to the root.

Sleator and Tarjan proved that by using this approach, all three basic operations (***find***(), ***insert***() and ***remove***()) have a logarithmic time bound. Another benefit of splaying is that the most frequently accessed items tend to gather near the root, therefore improving access speed, especially on skewed input sequences – the sequences, in which only a small number of items are accessed often while other items occur rarer. This property is exploited in the Splay eviction policy (Section 3.2.3).

Even though splay trees show several interesting theoretical properties, in practice they are outperformed by more conventional BSTs, like AVL or Red-Black tree[7] (the performance evaluation of NUMDB (Section 5.3) reaffirms this statement). This is due to the fact that in the splay tree the structure of the tree is altered on every operation, including find operation, while AVL, for instance, modifies the tree only during insertions and removals.

The typical use scenario for those data structures is a scenario, where a vast majority of operations is the search operation, while updates are not so often. AVL and Red-Black trees happen to be faster because they execute fewer instructions per find operation. Moreover, they do not make any writes to

memory during the lookup, and, as a consequence, there is lower load on the memory bus and the system cache.

Further researches on splay trees were focused in the main on how to reduce the number of rotations during splaying. An extensive overview of those optimizations is provided in [7]. One of the described techniques, the partial splaying is a modification of a conventional splay tree, where every node contains a counter that denotes a total count of accesses to this node. As usual, splaying is performed on every access, but the node is splayed only until its access count is less than the access count of its parent.

W. Klostermeyer showed that this modification does not gain any noticeable advantage over a standard splay tree [8]. However, partial splaying and other derived modifications can have some interesting properties specifically in application to a numerical database. It will be discussed in Chapter 3.

1.1.4 Hash Table

A hash table is another popular data structure that implements dictionary abstract data type. It uses the entirely different approach on item storage and lookup. A hash table allocates a contiguous array A, which size is bounded by the expected number of items to be stored, often multiplied by the *load factor* α. The items are stored in A.

Firstly, let's look at the simplified case: the key K that is used in a hash table is of an integer type. Having A, K and the value V, associated with K, it is possible to use a remainder of the division of K by the size of A as an index in A. Then, V will be stored in A at this index. This approach would give the best performance possible, as the V can be retrieved immediately and *the lookup time does not depend on the total count of items* in the hash table. However, since the *modulo* operation has been used, there can be several keys that point at the same index in A. This circumstance is called *collision*.

To deal with a collision, it is necessary to store K itself together with V, so that in case of a collision it would be possible to tell if the stored V is actually associated with the K or another K', that collides with K. Secondly, one must pick a strategy on how to deal with the case when two different keys, K_1 and K_2, that point at the same index are inserted. There are two main approaches:

Separate chaining (open hashing) – each element in A is a linked list (or another data structure), that stores all pairs $\langle K, V \rangle$ that collides.

Linear probing (closed hashing) – if during insertion of K in A at the index i a collision occurs (i is already occupied), a special function F is used to determine the second index at which K can be inserted. If it is also occupied, 3rd and all consequent positions, generated by F, are used to try to insert the element.

The approach described above can be generalized on keys of any type T. It is achieved with the help of a *hash function*. This function takes an argument of type T and maps it to an integer, called *hash*. This function must satisfy two properties:

Determinism – it should *always* map the same input to the same hash.

Uniformity – if used with a uniformly generated random sequence of objects on input, the hash function should produce a uniformly distributed sequence of hashes.

In [2, p. 464] Sedgewick and Wayne provide the detailed explanation of collision avoidance strategies as well as general information about hash tables. More information about hash function properties and hash function construction is presented in [9].

1.2 Thread Synchronization

1.2.1 Coarse-grained Locking

A trivial way to parallelize a sequential data structure is to eliminate a concurrent access at all. It can be achieved with a single mutual exclusion lock – *mutex*. While a thread holds a mutex, no other threads can lock the same mutex.

The sequential data structure is wrapped into the helper type, that locks the mutex in the beginning of every operation and releases it in the end, so that only one thread can access the data structure at a time, no matter how many threads are involved. This approach is called the *coarse-grained locking*, in contrast with the *fine-grained locking*, where many locks are used and each lock protects only a part of the data structure, so other threads can freely access other parts.

Pros of this approach is a very trivial implementation and the absence of any special requirements on the underlying data structure. However, coarse-grained locking is only suitable when a data structure has a support role in the program and is used occasionally. If the data structure is the key element

of the application, then a single lock becomes the bottleneck in the software, drastically decreasing program scalability. In this case one should use more sophisticated parallelization approaches.

1.2.2 Binning

The evolution of coarse-grained locking is the *binning*. The main drawback of the previous approach is that a single lock becomes the main point of contention between threads. One way to cope with this is to increase the number of locks. In contrast with the fine-grained locking, the binning does not involve any modifications of the underlying container.

Firstly, the numbers of bins – independent data structure instances – is chosen. Then a mapping between the item domain and a bin number is introduced. The mapping should yield a uniform distribution of mapped values. Every item is stored only in its assigned bin. Every bin has its own mutex, therefore the access to every bin is serialized. But since items are mapped uniformly, it is expected to produce much less contention than in case of a single lock.

1.2.3 Fine-grained Locking

The fine-grained locking usually offers better scalability, than the previously discussed approaches. Instead of a single lock, many mutexes are used simultaneously. Every mutex protects its part of data. The contention between threads is lower as it is unlikely that several threads will access the same portion of data at the same time.

However, this is true only if every portion has the same probability of being accessed (like in a concurrent hash table). In some data structures, typically binary trees, there are some nodes that are accessed (and are locked before access) oftener than others, e.g. the root in a binary tree.

Substantial modifications got to be made to the data structure to integrate the fine-grained locking. Sometimes the overhead added by this approach is so big, that it brings to naught any potential speed-up. Fine-grained locking is not a silver bullet, but usually it offers a reasonable trade-off between implementation complexity and application scalability.

Chapter 2

Numerical Database

In this chapter the contributions of [1] – the numerical database system and the wighted search tree – are explained in detail.

2.1 Concept

In the year 1990 S.C. Park, J.P. Draayer, and S.-Q. Zheng introduced a memoization system, called *numerical database*[1]. Like every memoization system, its primary goal is to reduce costly redundant calculation. The main idea behind this concept is to design a data structure that stores a limited number of items – key-value pairs – and provide an efficient way to retrieve, insert and remove items. A value is associated with the key that can be used to calculate the value by a function. The complete process of the item retrieval is demonstrated in Algorithm 2.

Algorithm 2 Numerical database item retrieval

```
1: procedure RETRIEVE(numdb, K)
2:     (V, found) ← numdb.find(K)
3:     if found = true then
4:         numdb.update_priority(K)        ▷ Can be embedded into find()
5:         return V
6:     end if
7:                         ▷ The item was not found and must be recalculated
8:     (V, priority) ← numdb.call_user_function(K)
9:     if numdb.is_full() then
10:        numdb.evict_item()   ▷ Remove the item with the lowest priority
11:    end if
12:    numdb.insert(K, V, priority)
13:    return V
14: end procedure
```

2.2 Priority

Each item has its assigned priority. When accessed, the priority of the node is updated. Park et. al. define priority as follows – "A good priority strategy should enable the frequency of a data item and its intrinsic value to be incorporated into its assigned priority." Initial priority can be supplied by an external algorithm or computed using some heuristics. For example, the heuristic can be based on the time it took to compute the value: some items can take much more time to be calculated than others, then these elements should be kept in the database even if they are accessed relatively rarely.

When a numerical database reaches its maximum capacity, the item with the lowest priority is removed from the database prior to the next insertion. This implies that the database should support queries on the item with the minimum priority. The way it is achieved in the original proposal is discussed in Section 2.3.

Park et. al. introduced a space-optimal representation of the item priority. To distinguish this representation from others, it will be called the *weighted search tree priority* (WST priority). This representation is stored in a single 32 bits long unsigned integer, but combines both the base priority and the hit frequency at the same time. The first 8 bits of the number are reserved for the base priority while remaining 24 bits contains hit frequency, multiplied by a base priority. Therefore, the actual priority value equals to

$$hit\_frequency \times base\_priority \times 256 + base\_priority \tag{2.1}$$

Another advantage of the WST priority is a simple adjustment procedure when the hit frequency is updated (see Algorithm 3). However, this representation also has some drawbacks that will be discussed in Section 3.1.1.

Algorithm 3 *WST* priority update

```
1: procedure UPDATE(priority)                    ▷ 4 bytes long unsigned integer
2:     base_priority ← priority & FF16                          ▷ Hexadecimal number
3:     priority ← priority + base_priority × 10016
4:                          ▷ Multiplication shifts the number 8 bits to the left
5:     return priority
6: end procedure
```

2.3 Weighted Search Tree

Park et. al. also proposed a data structure, called weighted search tree (*WST*), that can be used as a base for the numerical database. The weighted search tree is a combination of two well-known data structures – the AVL tree and the binary heap. Each of them fulfills its purpose:

AVL tree is used for a fast item lookup, insertion, and removal in $\mathcal{O}(\log \boldsymbol{size}(T))$ time.

Binary heap is used to maintain priorities of nodes. Specifically, it provides an ability to find a node with the lowest priority in $\mathcal{O}(1)$ and can perform insertions and deletions in $\mathcal{O}(\log \boldsymbol{size}(T))$ time.

Weighted search tree holds all its nodes in a single contiguous array. These nodes are ordered in the same way as in a regular minimal binary heap:

$$\begin{aligned} \forall N \in \textsc{Heap}, \boldsymbol{priority}(N) < \boldsymbol{priority}(\boldsymbol{heapLeft}(N)) \wedge \\ \boldsymbol{priority}(N) < \boldsymbol{priority}(\boldsymbol{heapRight}(N)) \end{aligned} \tag{2.2}$$

$\boldsymbol{heapLeft}(N)$ and $\boldsymbol{heapRight}(N)$ of N with index i are defined as $(2 \times i)$ and $(2 \times i + 1)$ respectively.

The difference is that each node is an AVL tree node at the same time – it stores links to its left child, right child, and parent node.

With a given structure the three basic operations are defined as follows:

find() is the same as in any binary tree – it is begins at the root and then continues as described in Algorithm 1.

insert() consists of two steps:

1. The node is inserted into the binary search tree, and the tree is adjusted using AVL rotations. Obviously, these rotations only change pointers inside nodes. Therefore, they do not affect the binary heap structure.

2. The node is inserted into the binary heap. It is done using the ***heapify***() operation[2, p. 346]. Note that heap adjustments would *reorder nodes* so that binary search tree pointers would point to wrong nodes and the whole *tree would become ill-formed*. To avoid this, special care should be taken when swapping nodes – one should also check and adjust tree pointers during swaps.

remove() is similar to the insertion, but steps are performed in the reverse order – at first, the element is removed from the heap, then from the tree. Again, heap adjustments should be performed with respect to the tree structure.

2.4 Known Implementations

The reference implementation was made by Park, Bahri, Draayer, and Zheng. A complete source code is available at [10]. It was implemented in Fortran programming language. The source code cannot be compiled with any modern Fortran compiler[11]. Therefore, this implementation is mentioned here but cannot participate in the benchmark. The NUMDB library contains the implementation of the weighted search tree, recreated basing on [1] and [12].

Another implementation has been developed and described by Miroslav Masat in his bachelor's thesis [11]. C++ source code is available at [13]. In this particular implementation item priorities are defined solely by the user and are not updated afterward. Therefore it is only applicable in scenarios when the item importance is known beforehand. A canonical numerical database has a broader field of applications.

Another essential flaw of Masat's implementation is that his weighted search tree operates on keys only, providing no facility for storing values, associated with keys[1].

[1]Practically, it is possible to overcome this limitation. Both the key and the value can be merged in a single structure, and the overloaded comparison operators would compare only keys. It is still unclear, why it was not implemented in the library itself.

What is more, *the key is stored twice for each item.* The AVL tree and the binary heap are not incorporated into the single data structure, but are kept completely separate. And each of them stores its own copy of keys. This fundamentally violates the original WST design.

Considering these flaws, Masat's work is excluded from the performance evaluation. It seems, that there is no advantage compared to the WST implementation, provided in NUMDB.

Both of the implementations are designed for a single-threaded environment only. It is a huge drawback since it is natural to run scientific applications on many-core systems. Therefore, a memoization system must support concurrent access in order to be usable in real-world applications. It was achieved in the NUMDB library.

Chapter 3

Numerical Database Variations

All of the main contributions of this thesis are presented in this chapter. Specifically, a variation of eviction policies, other than what have been proposed in [1], and alternative data structures, that can be used as the container for the numerical database, are discussed.

3.1 Priority

3.1.1 Improved WST Priority

The weighted search tree priority representation, discussed in Section 2.2, has some drawbacks that probably were irrelevant at the time when [1] was published. The problem is that keeping the hit counter in only 24 bits (even more, the hit counter multiplied by the base priority) would result in an integer overflow sooner or later.

For example, imagine the case when the base priority is the maximum possible – 255. Binary representation is $\texttt{000000FF}_{16}$. After 65793 hits, the priority will have its maximum value – $\texttt{FFFFFFFF}_{16}$. If another hit counter adjustment is made an overflow occurs, producing the value $\texttt{000000FE}_{16}$. Therefore, the maximum priority becomes very low, and even the base priority has been changed. There are at least two possible solutions:

Use larger counter – store the WST priority in a 64-bit integer. An overflow of a 56-bit counter is unlikely, not to say impossible – even with the maximum base priority, an overflow will occur only after the 282578800148737th insertion. It would take days to make so many adjustments, even if the processor performs only these adjustments and nothing else (which is at least impractical). Nevertheless, a certain disadvantage is that the memory overhead per each node is increased by 4 bytes.

Algorithm 4 *WST* priority update with saturation

```
1: procedure UpdateSaturated(priority)▷ 4 bytes long unsigned integer
2:     base_priority ← priority & FF16
3:     new_priority ← base_priority            ▷ 8 bytes long unsigned integer
4:     new_priority ← new_priority × 10016
5:                                  ▷ Maximum possible result is FFFFFFFF0016
6:     if new_priority < FFFFFFFF16 then      ▷ Maximum value for 4 bytes
7:         priority ← new_priority + base_priority
8:     else
9:         priority ← FFFFFF0016 + base_priority
10:    end if
11:    return priority
12: end procedure
```

Perform a saturated addition when adjusting the hit counter (Algorithm 4). The *saturated addition* is the addition that yields the expected result if no overflow occurs during the operation and the maximum number for the given operand size otherwise. The drawback of this method is the slightly increased computation time.

The difference between these two methods is the common trade-off between time and space. Since the total count of items that can be stored in a database with the limited memory available is the crucial characteristic of a database, the preference is given to the *saturated addition* method.

3.1.2 Priority Aging

The WST priority has one more drawback – it is suitable only for static input distributions – distributions, which mean remains constant during the execution. However, if the mean is known beforehand it is possible to construct a static optimal BST[5, p. 442], that will be more efficient than any dynamic lookup data structure.

A numerical database with the WST priority performs poorly on the time-varying distribution – a distribution which mean changes over time – while this type of distributions is more common in the real world applications. The problem arises from the fact that the priority does not reflect when an item was accessed for the last time.

The worst-case scenario is the following: an item is added to a database, then it is accessed frequently hence its priority rises to the maximum, and then is not used over a long time. During the runtime, several items like this can appear. Even though at some point they are not accessed anymore they are

still the most valuable items from the perspective of the database hence they will be kept much longer than other items. This pollutes the database with elements that are stored but not used.

There are several ways to cope with the problem. First of all, it is possible to use an entirely *different eviction policy*, the one that is not based on the item priority. Some of these policies are described in Section 3.2.

Another solution is to adjust a priority not only when the corresponding node is accessed but also when it is visited (during the lookup of another item). For example, when searching in a binary search tree, the priority of the node N that is being searched is increased while priorities of the nodes, lying on the path between the root of the tree and N, are degraded.

This mechanism may not be effective with the AVL tree because in the AVL tree the order of the nodes does not correlate with node priorities. However, it seems very promising in application to the splay tree – by applying this mechanism, the nodes near the root can stay there if only they are constantly accessed.

3.2 Alternative Eviction Policies

The canonical weighted search tree always chooses the node with the minimum priority for the deletion. However, it is only one of many possible eviction policy. Some other policies are presented in this section. From general ones, like LRU, to those exploiting the lookup data structure internals to find the least valuable item.

3.2.1 LRU Policy

The Least-Recently-Used policy tracks every access to the items and sorts them by the access order. Then it evicts the item that was not accessed for the longest time. The common implementation is based on a doubly-linked list. When an item is accessed its corresponding node in the LRU list is moved into the head of the list. Then the least-recently-used node is the one in the tail of the list. When a new item is added, it is inserted in the head of the LRU list.

3.2.2 LFU Policy

The Least-Recently-Used item policy fulfills the same purposes as the LRU policy. But when it decides which item should be evicted, the access frequency is also taken in account in addition to the last access time (LRU uses the latter property only).

3.2.3 Splay Policy

A splay tree tends to keep the most frequently accessed items near its root. By relying on this property, it is possible to eliminate a separate data structure that manages item priorities. When an eviction is performed, one of the bottom nodes is chosen for eviction. Even though this strategy may not choose the optimal node every time, it is expected to perform effectively on average. Moreover, this approach has the lowest memory overhead per node among all tested data structures.

3.3 Alternative Sequential Containers

3.3.1 Hash Table

One of the data structures that can be used in place of the weighted search tree is the hash table. Hash tables have faster than balanced BSTs lookup time under most workloads. What is more, a node in a hash table has lower memory overhead than a binary tree – with open hashing (based on a doubly linked list) every node stores only 2 pointers compared to 3 in a binary tree node and using closed hashing implies that no pointers are stored at all.

However closed hashing can not be used because when the hash table is almost full, a lot of unsuccessful probes occur before a suitable index is found. Usually, this problem is solved by rehashing – if the count of probes exceeds the certain limit, the hash table is expanded. However, it is impossible in the numerical database since the amount of available memory is preset and cannot be exceeded.

On the other hand, limited memory is rather an advantage for the open hashing. If the total amount of memory available is known beforehand, then a hash table with open hashing can be preallocated to its maximum size and be never rehashed after. This, in turn, allows a concurrent version of a hash table to be simplified as the concurrent rehashing is one of the hardest problems to cope with.

3.3.2 Splay Tree

Another data structure that looks promising is the splay tree. As it was mentioned in Section 1.1.3, usually splay trees tend to be slower than AVL. However, in application to the numerical database, it is possible to exploit the fact that the least valuable nodes are usually gathered in leaves of the tree. Therefore it is is possible to eliminate a binary heap from a numerical database and to use the splay policy, as described in Section 3.2.3. What is more, it is possible to implement a concurrent numerical database using the concurrent splay tree[14].

3.4 Alternative Concurrent Containers

3.4.1 Coarse-grained Lock Adapter

The NUMDB library provides a universal adapter, that wraps a sequential container and adapts it to the concurrent environment by using the coarse-grained locking approach (Section 1.2.1). It has the same interface, as a usual numerical database container. All methods follow the same structure:

1. the mutex is locked
2. the call is forwarded to the underlying container
3. the mutex is released

3.4.2 Binning Adapter

The binning adapter class realises the binning concept (described in Section 1.2.2). It is similar to the coarse-grained lock adapter, however, it encapsulates several instances of a container, each with own mutex.

The number of bins is passed in the class constructor. The mapping is defined as $\boldsymbol{hash}(K) \bmod bin\_count$. Every bin is represented by a sequential container, e.g. the weighted search tree. In order to preserve the memory limit, all available memory is equally divided between all bins.

3.4.3 CNDC

For the purposes of the NUMDB library, the original concurrent container, called *Concurrent Numerical Database Container* – CNDC, has been developed. It defines 3 thread-safe operations – ***find***(), ***insert***(), and ***removeMin***(). Thread-safeness is achieved through the fine-grained locking approach. CNDC is based on concurrent versions of the hash table and the binary heap. Sequential benchmarks (Section 5.3) proved that the combination of a hash table and a binary heap outperforms numerical databases, that are based on the LRU and LFU eviction policies.

Fine-grained locking hash table implementation is much simpler compared to a similar concurrent BST. A lock is assigned to every hash table bucket (or every k buckets) and every operation inside the bucket locks the corresponding mutex. Since an operation in one bucket never interferes with any other bucket, only one lock is needed per operation, while other threads can operate on other buckets at the same time. Therefore, the overhead added by locking is smaller, than in a concurrent BST, where up to $\log \boldsymbol{height}(T)$ mutexes has to be locked on every operation.

There are several known binary heaps with a fine-grained locking ([15], [16] and more). The CNDC is based on the CHAMP binary heap, developed by Tamir, Morrison, and Rinetzky [16]. Unlike the majority of concurrent binary heaps, CHAMP allows priorities to be updated after the insertion.

In the following section, two types of locks are distinguished – the *bucket* mutex (the one, that protects a single bucket in a hash table) and the *heap* mutex (the one, that protects a single item in a binary heap. Every hash table node has a link to the corresponding heap node and vice versa. CNDC operations are defined as follows:

find() consists of the following steps. At first, calling thread locks the corresponding bucket mutex. The requested item is searched in the bucket.

If the item is found, its priority is updated. Before updating the priority the corresponding heap mutex must be locked. After locking the heap mutex the link to the heap node is checked again. If it has changed, the heap lock is released and the operation is repeated. Double check is required, because another thread can change the link even in case it does not hold the bucket mutex.

However, the heap mutex is required to be locked prior to the link update. Therefore, when a thread holds a heap mutex it is guaranteed, that no other thread can change the link between the heap node and the hash table node.

When the node is locked, the priority is updated and the ***bubbleDown***() operation (as defined in [16]) is performed. ***bubbleDown***() internally releases the heap and bucket locks.

insert() has a structure, similar to ***find***(). The corresponding bucket mutex is locked. New item is inserted into the hash table. After that, the item is inserted in the binary heap (at the last index). Before the insertion is performed, the heap lock of the last index is locked.

The bucket mutex is released. It is possible to do this so early, because the *heap* lock will be held till the end of the operation. While it is locked, no other thread can execute any operation on the same node.

Finally, ***bubbleUp***()[16] is performed. It propagates the node down until the heap invariant is restored.

removeMin() evicts the item with the lowest priority. This operation is decomposed into three independent parts.

1. The item is evicted from the heap.
2. The hash table node is marked as *deleted*. Otherwise, another thread can access the item and start the priority update routine. Since the node does not exist in the heap anymore, the thread will enter into an infinite loop. Marking solves the problem as follows – the marked node can still be accessed by other threads, however, they will skip the priority update stage for the node.
3. The item is removed from the hash table.

Chapter 4

Library Implementation

In the following chapter the NUMDB library is described. This library provides the implementation of several containers (including concurrent containers), discussed in Chapters 2 and 3.

At first, global design decisions are explained. Then the most important classes are analyzed and the particular tricks and code optimizations used in the implementation are described.

4.1 Chosen Technologies

The library is written in the C++ programming language. It has been chosen by the following criteria:

- C++ is a compiled language. Dynamic and managed languages have a huge runtime overhead, that significantly affects general application performance, while not providing any serious advantage (at least from the perspective of scientific applications).
- Modern C++ compilers applies lots of advanced code optimizations, increasing the performance gap between compiled and dynamic languages even further.
- C++ has an advanced template system that helps to write highly reusable and extendable code, while adding no overhead in runtime.

The library uses some language and standard library features that have been introduced in the C++14 standard. Therefore, the compiler must be C++14 conformant.

The build process is managed by the `CMake` software, one of the most popular tool in this category. A great number of other projects also use `CMake` for managing the build process. It is trivially to embed a one CMake-based project into another.

`CMake` uses `CMakeLists.txt` file, that contains the project definition, to generate a make file (other generators are also supported). Then, a program or a library can be built by invocating the GNU `make` utility.

4.2 Project Structure Overview

4.2.1 Library

The project consists of the following folders:

`include/numdb` folder containing header files

`lib` folder with source files (that can be compiled separately and merged with the user program during the linkage stage)

`test` folder containing unit tests

`benchmark` folder with the benchmarking program, that has been used to perform performance evaluation (Chapter 5)

Since the library heavily relies on templates, the majority of code is placed in the header files.

4.2.2 External Libraries

This library relies on several additional libraries. They are distributed along with the sources (except Boost.Math) in the `3rdparty` folder in the form of the git submodules. Since all of them use `CMake` software for building process, they are natively integrated into the main project and are compiled automatically when needed.

function_traits extends C++ standard library metaprogramming capabilities by defining the type trait that can *deduce argument types* of a provided functor object.

murmurhash2functor library contains `murmurhash2` hash function[17] implementation and wraps it into the interface, similar to the `std::hash`.

The demand for the `std::hash` replacement is dictated by the fact that the standard hash is not suitable for the hash table –

`std::hash` for an integer is defined as the number itself (at least on some compilers[18]); under certain circumstances, this can lead to a high number of hash collisions.

GOOGLE BENCHMARK framework is used as a benchmark starter. Its responsibility is to run functions that are to be benchmarked, measure their running time and other metrics, and encode the result into a structured data format (e.g. `JSON`).

GOOGLE TEST framework enables unit testing. It provides some helper function and macros to simplify writing of unit tests as well as a common facility to run and evaluate tests.

BOOST.MATH provides some statistical functions that are required by the benchmark program. This is the only library that is not bundled with the project and must be installed separately.

FUNCTION_TRAITS and MURMURHASH2FUNCTOR are required by the library itself. Other libraries are needed for testing/benchmarking only.

4.3 Source Code Overview

In the following section individual parts of the library are analyzed. The numerical database is called *function cache* in this implementation because it has a cleaner meaning than the less known numerical database term.

All classes, presented in NUMDB, are declared in the `numdb` namespace.

4.3.1 numdb.h

This is the main entry point of the NUMDB library. Users need to include this file into their project to gain access to the data structures provided by the library.

4.3.2 function_cache.h

`FunctionCache` is the main class that realizes the numerical database concept. It does not determine how items are stored – it takes a container as a template type parameter and stores all items in the instance of the provided container.

This class defines helper function and type definitions, that are common for all numerical database implementations, e.g. `args_tuple_t` – a tuple that can store arguments of a function call. What is more, the item retrieval operation, as described in Algorithm 2, is implemented with actual calls to the lookup and insertion routines forwarded to the underlying container.

`FunctionCache` is responsible for calling the provided user function in case the requested item was not found in the container. All invocations are timed with a system clock; then the duration is converted into the initial priority that is assigned to new item (initial priority generator is used).

4.3.3 initial_priority_generator.h

This header file contains classes that are responsble for computing the initial item priority basing on the duration of the current user function call and durations of previous calls.

`MinMaxPriorityGenerator` calculates the priority as a linear interpolation between the minimum and the maximum values. `RatioPriorityGenerator` calculates it as a proportion to the current average value. The latter scheme proved to be fairer and is used in the final implementation.

What is more, both schemes have an adaption mechanism – the average is divided by 2 every N iterations, so that the latest input has bigger influence on the final result than data from previous periods. At the same time, the historic data is not discarded completely.

4.3.4 fair_lru.h and fair_lru.cpp

`FairLRU` class implements the alternative eviction policy – the item accessed the least recently among all items is always chosen for eviction. It is achieved by maintaining a doubly linked list of all nodes. When a node is inserted, it is placed in the tail (end) of the list. When a node is accessed, it is extracted from its current position, then inserted in the tail of the list. Therefore, the least recently used node appears in the head (start) of the list. All mentioned operations – insertion at the end, extraction (with a known pointer to a node), extraction from the head – have $\mathcal{O}(1)$ time complexity.

To embed `FairLRU` in a container, an instance of `FairLRU` should be added as a class member and container nodes should be derived from the `FairLRU::Node` class (it contains data members that are required for a doubly linked list implementation). Then all basic operations, namely ***find***(), ***insert***(), and ***remove***(), should call corresponding methods on the `FairLRU` instance.

4.3.5 fair_lfu.h and fair_lfu.cpp

`FairLFU` external interface and usage scenario are similar to the ones in `FairLRU`. However, it differs in the way it choses items for eviction (see Section 3.2.2).

Internally, the implementation is based on the two-level linked list, as described by Shah, Mitra, and Matani[19]. This implementation has been chosen because it guarantees $\mathcal{O}(1)$ time complexity on all basic operations. Another well-known LFU implementation is based on a binary heap, but it achieves only $\mathcal{O}(\log N)$ time complexity.

The implementation used in the library differs from the original one. When a new item is inserted, the original implementation always assigns 1 to the node hit count. This yields a very ineffective behavior as the LFU tends to evict nodes that have just been added and preserves older ones, even those that have been accessed only twice.

The solution of this problem, presented in the library, is to calculate the initial hit counter value as a hit count of the least frequently accessed node (the one that is in the list head and may be evicted in the next step) incremented by one. This approach ensures that a new node is never inserted at the list head.

As a side effect, this adds the priority aging process (the approach is inverted – priority of new items is boosted instead of decreasing priority of older ones).

4.3.6 weighted_search_tree.h

Basing on [1] and [12], the original weighted search tree has been reimplemented. Unlike the original implementation, the priority aging (see Section 3.1.2) is also implemented – while traversing over the AVL tree (during item lookup), priorities of all visited nodes are decreased, and the binary heap is adjusted accordingly.

The improved WST priority scheme (as discussed in Section 3.1.1) is used for the node priority representation. Another optimization is the elimination of the AVL balance factor as a separate structure member (in this case it takes at least 1 byte). It is embedded into the priority component – 2 bits are reserved for the balance factor, and remaining 30 are used for the priority (8 bits for the base priority and 22 bits for the accumulated priority). It is achieved with the C++ *bit field* feature.

4.3.7 hash_table/

`hash_table` folder contains `FixedHashtableBase`, `FixedHashtableBinaryHeap`, and `FixedHashtableFairLU` classes.

`FixedHashtableBase` defines hash table implementation, that is common for all derived classes. However, it contains no logic for choosing a node to be evicted – `FixedHashtableBase` forwards the call to its derived class, where this operation is implemented.

Listing 1 Curiously Recurring Template Pattern

```
template <typename DerivedClass>
struct Base {
    void callFoo() {
        static_cast<DerivedClass*>(this)->foo();
    }
};
struct Derived : public Base<Derived> {
    void foo() {
        std::cout << "Derived foo" << std::endl;
    }
};

int main() {
    Derived d;
    d.callFoo(); //prints "Derived foo"
    return 0;
}
```

Normally, this polimorhic behavior can be achieved with a virtual method call. However, virtual invocation brings additional overhead. Another drawback is that a virtual call can not be inlined by a compiler. Note, that we are dealing with the static polymorphism – all functions that can be called are known at compile time and are never changed in runtime, in contrast to the dynamic polymorphism.

To simulate the static polymorphism, *Curiously Recurring Template Pattern* is often used[20]. The base class (that needs to call a function which implementation is provided only in the derived class) takes its derived class as a template argument. When a polymorphic method needs to be called, base class object casts `this` pointer to the derived class and then calls the desired function. Name lookup mechanism finds the implementation of the method in the derived class and performs a call to it (see Listing 1). It is a regular call, so a compiler can apply call inlining and other optimizations.

Two other classes, `FixedHashtableFairLu` and `FixedHashtableBinaryHeap`, derive `FixedHashtableBase` and supply the implementation of the method, that searches for the least valuable item, that is to be evicted. To choose the node, `FixedHashtableFairLu` uses `FairLRU` or `FairLFU` manager (actually, the manager is passed as a template parameter, so it is possible to extend the implementation with a custom manager). `FixedHashtableBinaryHeap` maintains a binary heap for item priorities. In fact, it very very similar to the weighted search tree, but with the hash table used in place of the AVL tree.

4.3.8 splay_tree/

`splay_tree` folder contains different variations of the splay tree. Practically the complete tree implementation is in the `SplayTreeBase` class. Similarly to the hash table implementation, the code for determining the least valuable item is excluded from the base class. Again, the derived classes are responsible for implementing it.

This splay tree implementation does not rely on the parent pointer in any way. Therefore, it could be excluded from the node declaration. This would decrease the memory overhead per node. However, for some item eviction policies (LRU and LFU), a pointer to the parent is an inevitable requirement (for other policies it is not). To support both types of policies, a wrapper over a parent pointer is introduced. The wrapper interface consists of get and set operations. Two implementations of the wrapper are provided – an actual implementation, that encapsulates a real pointer, and a mock one, that stores no value.

Derived classes tell `SplayTreeBase` (through a trait class) which wrapper implementation they require and `SplayTreeBase` embeds the chosen wrapper into the `SplayTreeBase::Node` structure.

Even though the mock wrapper contains no data members, the size of an empty structure can not be zero in C++[21]. Therefore, if the wrapper is embedded into a node, it takes at least one byte while not containing any valuable information. This unnecessary overhead is eliminated using the *Empty Member Optimization*[22].

There are two derived classes, that are responsible for the item eviction policy:

`SplayTreeFairLu` is similar to the `HashTableFairLu` class – it reuses LRU and LFU node eviction policies.

`SplayTreeBottomNode` realizes the Splay eviction policy, as described in Section 3.2.3.

4.3.9 Concurrent containers

NUMDB contains 2 concurrent adapters, namely `CoarseLockConcurrentAdapter` and `BinningConcurrentAdapter`, and CNDC data structure implementation, described in Section 3.4.

Chapter 5

Performance Evaluation

This chapter describes the benchmarking program, its input parameters, and the specification of the machines, that were used for the performance evaluation. Additionally, the results of the benchmark runs are analyzed (the results itself are presented in Appendix B) .

5.1 Benchmark

The performance evaluations is done with the benchmarking program. For each tested data structure (presented in Chapter 2 and Chapter 3), the benchmark runs for a specified amount of time. Then the throughput (operations per second) is calculated as the total count of iterations divided by the total time.

Instead of separately measuring the performance of every basic operation (***find***(), ***insert***(), ***remove***()), the overall numeric database performance is evaluated. The numeric database retrieval operation consists of either lookup (in case the item with the specified key is in the database) or lookup, user function invocation, item removal and item insertion. The user function is much slower than the database operations. Therefore, the most effective numerical database is the one that calls the user function as rarely as possible.

The recursive computation of the Nth Fibonacci number is chosen as the user function. The algorithm implementation is trivia and, having the exponential time complexity, it is extremely inefficient. This is the advantage from the perspective of the benchmark as its task is to simulate a very computational-heavy function. What is more, the time it takes to compute the function can be easily adjusted by the function argument.

Generally, cache systems perform well only on skewed inputs, when there is a small subset of items that are accessed most of the time while other items are accessed much less often. In this benchmark, a numerical database is tested with a random input sequence that has normal distribution of values. The parameters of the distribution are as follows:

Mean u equals zero. The data structures presented in this library are agnostic to the particular argument values and their performance is only affected by the frequency of items in the input sequence. So u can be any number. Zero is any number.

Standard deviation σ is derived from *Area-under-curve* parameter and the available memory. *Capacity* is calculated as the available memory divided by the size of a single item. *Area-under-curve* is a value from the interval $(0, 1)$. It defines the ratio of accesses to the *Capacity* most valuable items to the total count of accesses. With given *Capacity* and *Area-under-curve*, σ is calculated as follows:

$$\sigma = \frac{Capacity}{2 \times Quantile(0.5 + \frac{Area\text{-}under\text{-}curve}{2})} \tag{5.1}$$

5.2 Benchmark Parameters

The benchmark has several input parameters:

minval, maxval – the range of values the Fibonacci function is called with. It is defined as a range to simulate functions which execution time depends on its arguments.

available memory – the maximum amount of memory the numerical database uses.

thread count – relevant for concurrent numerical databases. Sets the number of threads to run in parallel.

mean changing rate – adjusts the rate at which the mean of the distribution is changed. If larger than zero, it simulates input sequences with non-static distribution. It is measured in the *delta-per-iteration* units – its value is added to the mean at every iteration, e.g. if the rate is $^1/_{100}$, than after 1000 iteration the mean will move by 10.

area-under-curve – the parameter that affects the standard deviation of the distribution.

5.3 Analysis of the Sequential Containers

The performance evaluation of the sequential containers has been done on the following machine:

CPU Intel® Core™ i5-6200U 2.30GHz (2.80 GHz[2])× 2 cores

Memory 8 GB DDR3 1600 MHz

OS Linux® Ubuntu® 16.04 LTS 64-bit

Compiler GCC 5.4, compilation flags: `-O3 -std=c++14`

In this chapter there are several graphs (Figure 5.1, Figure 5.2, Figure 5.3), that visualises data presented in Appendix B. The graphs do not include all the data from the tables, e.g. there is no graph for LRU and LFU based numerical databases; only the most representative candidates have been chosen. The outcomes have confirmed some assumptions, stated in the previous chapters, while refuted others.

The weighted search tree proved to be the most effective container for the numerical database. However, no concurrent WST implementation is known, so its usage is limited to the sequential environment.

The combination of the hash table and the binary heap performs about 10% slower than WST. However, its advantage over WST is the simpler structure, that made it possible to develop the concurrent version of this container, called CNDC.

The benchmark proved splay tree inefficiency. It is outperformed by WST and the hash table on all workloads. Therefore it is not recommended to use it for a numerical database.

Least-recently-used and *Least-frequently-used* eviction policies do not achieve the same performance as the policy, based on the binary heap. Possibly, because they do not rely on the initial item priority, thus give no preference to items that are rarely accessed but took long to be calculated and due to that must be kept in the database.

Priority aging does not show any advantage over the static priority. However, further studies and tests are required here, as the benchmark does not simulate the worst-case input for the static priority scheme.

[2]with Intel® Turbo Boost Technology

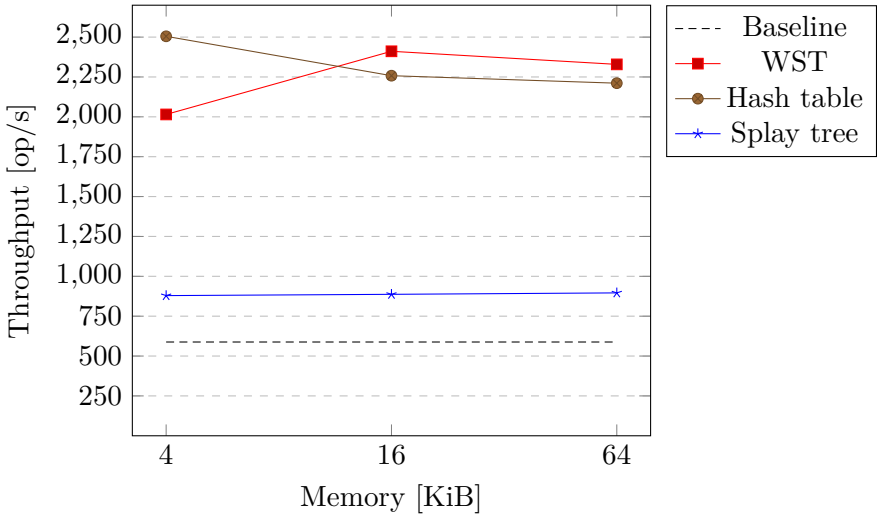

Figure 5.1: Sequential containers comparison (variable memory size)

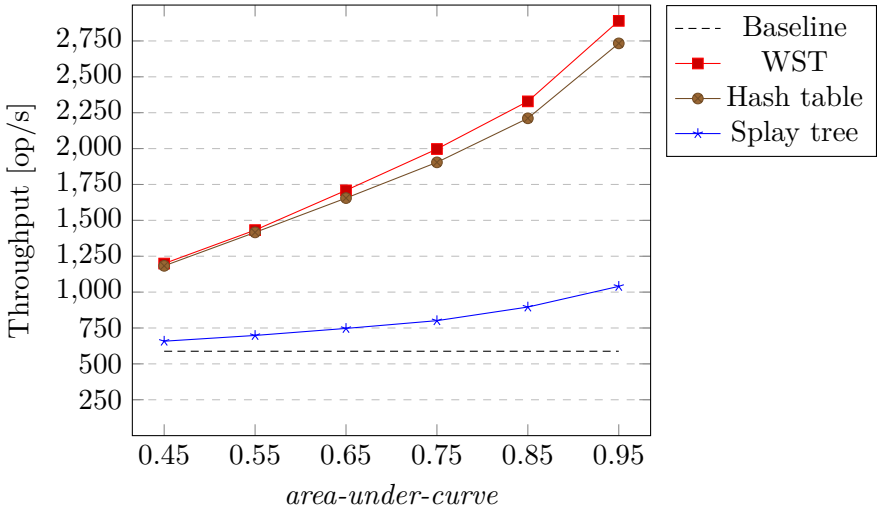

Figure 5.2: Sequential containers comparison (variable *area-under-curve* value)

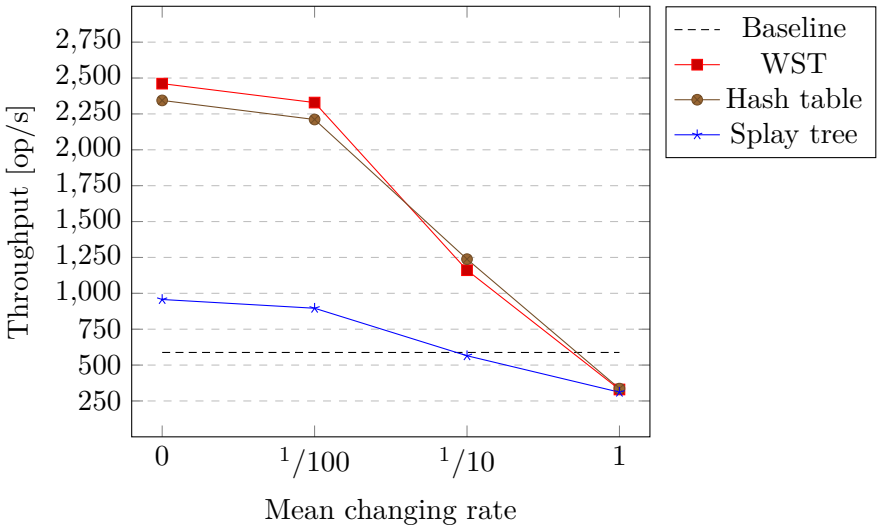

Figure 5.3: Sequential containers comparison (variable mean changing rate)

5.4 Analysis of the Concurrent Containers

The performance evaluation of the concurrent containers has been performed on the following machine:

CPU Intel® Xeon® E5-2650 v4 2.20GHz (2.90 GHz[3]) × 2 sockets × 12 cores

Memory 256 GB DDR4 2400 MHz

OS Linux® Ubuntu® 16.04 LTS 64-bit

Compiler GCC 5.4, compilation flags: `-O3 -std=c++14`

Same as with the sequential containers, the complete benchmark data is presented in Appendix B. Figure 5.4 shows the performance difference between the tested concurrent containers. Figure 5.5 demonstrates their scalability – the relative per-thread performance with increasing number of threads. For example, the graph shows that with 24 threads running simultaneously, the performance of one thread in cndc is 30% lower than that in the single-threaded test.

As expected, the coarse-grained locking approach yields the very ineffective container. The observation that with 2 threads the performance is two times lower compared to the single-threaded test (note, the total, not the per-thread performance) can be explained by the high contantion on the single mutex. The process of locking/unlocking an open mutex is quite fast operation, but locking the already locked mutex implies, that the thread will be suspended and then resumed. This is quite heavy and complex operation from the perspective of an operational system.

The binning approach performs better than the previous one. However, Figure 5.5 clearly shows, that it does not scale very good for the bigger number of threads – at 24 threads, per-thread performance is only 10% as high as the maximum.

cndc shows excellent scalability, even with 24 threads. The poor per-thread performance is rather explained by the huge memory overhead per item than by computational complexity – in addition to the overhead, introduced by the data structure itself, synchronization adds at least 60 bytes for each item. The further studies should be directed at lowering this overhead. For example, some synchronization can be performed in the lock-free fashion with no mutexes involved.

[3]with Intel® Turbo Boost Technology

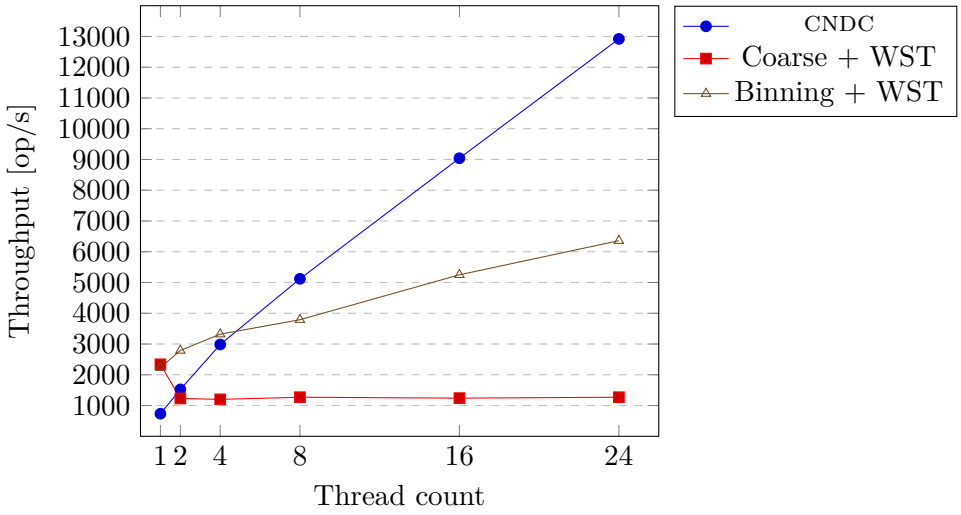

Figure 5.4: Concurrent containers comparison (variable thread count)

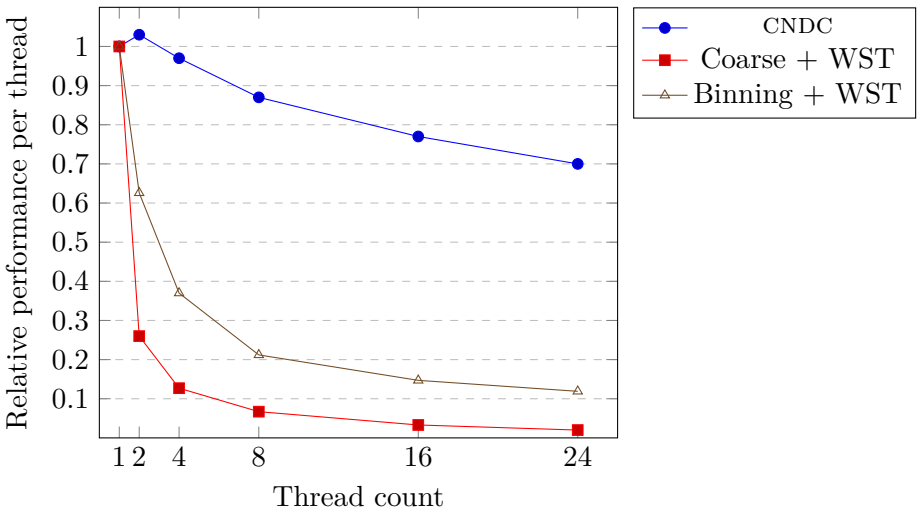

Figure 5.5: Concurrent containers scalability comparison

Conclusion

This thesis explores and extends the concept of the numerical database. First of all, an extensive overview of the original concept and its practical implementation is provided, basing on the [1]. In Chapter 3 the flaws of the original WST implementation are pointed out and the fix for them is presented.

In addition, the alternative data structures – the hash table and the splay tree, that can be used in place of WST, are discussed. The practical performance evaluation, presented in Chapter 5, showed that the hash table achieves almost the same performance as WST. However, the splay tree is much slower than other data structures.

Chapter 3 introduces several alternative policies for managing item priorities – LRU and LFU policies. However, the benchmark showed that the original policy, based on the binary heap, is more effective.

This thesis presents new concurrent data structure – CNDC. It is based on the combination of the hash table and the binary heap. It has been adapted to the multi-threaded environment using the fine-grained locking approach (Section 1.2.3). Its advantage is the high scalability with increasing number of threads. However, it shows rather low single-thread performance – this is due to the high memory overhead per node. Further researches, directed at lowering the overhead, should be done.

Finally, the NUMDB library is presented. It realises the data structures, mentioned above. The implementation is done in the C++ language. NUMDB has been used for the practical performance evaluation, presented in Chapter 5. On the particular test, the following speed up has been achieved:

Base algorithm without numerical database – 588 operations per second

Splay tree based numerical database – 976 operations per second

Hash table based numerical database – 2407 operations per second

WST based numerical database – 2595 operations per second

Bibliography

[1] Park, S. C.; Draayer, J. P.; et al. Time-Space Optimal Numerical Database for Large-Scale Scientific Applications. *Proc. Int. Computer Symposium*, 1990: pp. 333–338, visited on 2017-05-03. Available from: http://www.phys.lsu.edu/draayerpubs/ConferenceProceedings/Time-SpaceOptimalNumericalDatabaseforLarge-ScaleScientificApplications.pdf

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

[3] Black, P. E. complete binary tree. 2016, visited on 2017-05-03. Available from: https://xlinux.nist.gov/dads/HTML/completeBinaryTree.html

[4] Adelson-Velskii, G. M.; Landis, E. M. An algorithm for organization of information. *Dokl. Akad. Nauk SSSR*, volume 146, 1962: pp. 263–266, ISSN 0002-3264.

[5] Knuth, D. E. *The Art of Computer Programming, Volume 3: (2Nd Ed.) Sorting and Searching*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1998, ISBN 0-201-89685-0.

[6] Sleator, D. D.; Tarjan, R. E. Self-adjusting Binary Search Trees. *Journal of the ACM*, volume 32, no. 3, 1985: pp. 652–686, ISSN 0004-5411, doi:10.1145/3828.3835. Available from: http://doi.acm.org/10.1145/3828.3835

[7] Huus, E. Reduced Restructuring in Splay Trees. 2014, visited on 2017-05-03. Available from: https://eapache.github.io/assets/Huus2014_SplayTrees.pdf

[8] Klostermeyer, W. F. Optimizing searching with self-adjusting trees. *J. Information and Optimization Sciences*, volume 13, no. 1, jan 1992: pp. 85–95, ISSN 0252-2667, doi:10.1080/02522667.1992.10699094. Available from: http://www.tandfonline.com/doi/abs/10.1080/02522667.1992.10699094

[9] Knott, G. D. Hashing functions. *The Computer Journal*, volume 18, no. 3, 1975: pp. 265–278, ISSN 0010-4620, doi:10.1093/comjnl/18.3.265. Available from: http://comjnl.oxfordjournals.org/content/18/3/265.short

[10] Park, S. C.; Draayer, J. P.; et al. WSTREE. 1994, visited on 2017-05-03. Available from: http://cpc.cs.qub.ac.uk/summaries/ACTZ_v1_0.html

[11] Mašat, M. Numerical Database System. Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2016. Available from: https://dspace.cvut.cz/handle/10467/65871

[12] Park, S. C.; 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, ISSN 00104655, doi:10.1016/0010-4655(94)90172-4.

[13] Mašat, M. CCherish. 2016, visited on 2017-05-03. Available from: https://github.com/tyrhus/CCherish

[14] Korenfeld, B. CBTree: A practical concurrent self-adjusting search tree. 2012, visited on 2017-05-03. Available from: http://link.springer.com/chapter/10.1007/978-3-642-33651-5_1

[15] Hunt, G. C.; Michael, M. M.; et al. An Efficient Algorithm for Concurrent Priority Queue Heaps. *Information Processing Letters*, volume 60, 1996: pp. 151–157. Available from: http://www.research.ibm.com/people/m/michael/ipl-1996.pdf

[16] Tamir, O.; Morrison, A.; et al. A Heap-Based Concurrent Priority Queue with Mutable Priorities for Faster Parallel Algorithms. *19th International Conference On Principles Of Distributed Systems*, volume i, 1998: pp. 1–16, doi:10.4230/LIPIcs.OPODIS.2015.186. Available from: http://www.cs.tau.ac.il/~mad/publications/opodis2015-heap.pdf

[17] Appleby, A. MurmurHash. Visited on 2017-05-03. Available from: https://github.com/aappleby/smhasher

[18] Free Software Foundation, I. libstdc++: functional_hash.h. Visited on 2017-05-03. Available from: https://gcc.gnu.org/onlinedocs/gcc-6.2.0/libstdc++/api/a01298_source.html

[19] Shah, K.; Mitra, A.; et al. An (1) algorithm for implementing the LFU cache eviction scheme. 2010, visited on 2017-05-03. Available from: `http://dhruvbird.com/lfu.pdf`

[20] Coplien, J. Curiously Recurring Template Patterns. *C++ Report*, volume 1, no. February, 1995: pp. 24–27, ISSN 1040-6042.

[21] Stroustrup, B. Stroustrup: C++ Style and Technique FAQ. 2013, visited on 2017-05-03. Available from: `http://www.stroustrup.com/bs_faq2.html#sizeof-empty`

[22] Myers, N. C. The "Empty Member" C++ Optimization. *Dr. Dobb's Journal*, volume 1, no. C++ Issue, 1997, visited on 2017-05-03. Available from: `http://www.cantrip.org/emptyopt.html`

Appendix A

Container Reference

A.1 Sequential Containers

Baseline The performance of the user function itself without any numerical database.

WST, aging = 0 The original weighted search tree, as described in [12].

WST, aging = N Weighted search tree with aging mechanism. For every traversed node, its priority is decreased by $N \times 256$

Hash table, Binary heap, aging = N The container, very similar to the WST, but with a hash table in place of a BST. N parameter has the same meaning as for the WST; 0 value means that no priority aging is performed.

Hash table, LRU The hash table, that uses LRU strategy for item eviction.

Hash table, LFU The hash table, that uses LFU strategy for item eviction.

Splay tree, Splay policy, canonical The splay tree with the splay policy (as described in Section 3.2.3). "Canonical" stands for the splaying strategy – the node is always splayed up to the root[6].

Splay tree, Splay policy, partial The splay tree with the splay policy (as described in Section 3.2.3). "Partial" stands for the splaying strategy – each node has the access counter and it is splayed until its counter value is less than that of its parent the node. This approach is known as the *partial splaying*[8].

Splay tree, LRU, canonical The splay tree with the LRU policy.

Splay tree, LFU, canonical The splay tree with the LFU policy.

A.2 Concurrent Containers

The number after the container name in Table B.5 and Table B.6 denotes the number of threads running in parallel.

CNDC The *concurrent numerical database container*.

Coarse lock + WST The sequential WST container, guarded by a coarse-grained lock.

Binning + WST The sequential WST container, guarded with the binning approach.

Appendix B

Benchmark Results

Table B.1: Sequential benchmark (variable memory size, static input distribution)

Container	Throughput [op/s]		
	4 KiB	16 KiB	64 KiB
Baseline			588
WST, aging = 0	2265	2595	2461
WST, aging = 1	2502	2568	2279
WST, aging = 2	2619	2369	2052
WST, aging = 4	2651	2040	1699
WST, aging = 16	1500	1741	1721
Hash table, Binary heap, aging = 0	2716	2407	2344
Hash table, Binary heap, aging = 1	2686	2388	2353
Hash table, Binary heap, aging = 2	2640	2422	2391
Hash table, Binary heap, aging = 4	2547	2456	2335
Hash table, Binary heap, aging = 16	2568	2344	2316
Hash table, LRU	803	779	771
Hash table, LFU	750	711	706
Splay tree, Splay policy, canonical	1014	976	957
Splay tree, Splay policy, partial	766	963	823
Splay tree, LRU, canonical	721	700	679
Splay tree, LFU, canonical	671	663	659

Parameter	Value
Min/Max arg	25/35
Memory	*vary*
Area-under-curve	0.85
Mean changing rate	0

Table B.2: Sequential benchmark (variable memory size, time-varying input distribution)

Container	Throughput [op/s]		
	4 KiB	16 KiB	64 KiB
Baseline			588
WST, aging = 0	2015	2411	2329
WST, aging = 1	2175	2348	2183
WST, aging = 2	2368	2265	1931
WST, aging = 4	2430	1864	1609
WST, aging = 16	1343	1514	1560
Hash table, Binary heap, aging = 0	2505	2258	2211
Hash table, Binary heap, aging = 1	2320	2245	2220
Hash table, Binary heap, aging = 2	2295	2176	2199
Hash table, Binary heap, aging = 4	2354	2202	2204
Hash table, Binary heap, aging = 16	2121	2167	2152
Hash table, LRU	697	711	720
Hash table, LFU	639	646	651
Splay tree, Splay policy, canonical	879	887	896
Splay tree, Splay policy, partial	812	872	799
Splay tree, LRU, canonical	640	634	631
Splay tree, LFU, canonical	610	626	628

Parameter	Value
Min/Max arg	25/35
Memory	*vary*
Area-under-curve	0.85
Mean changing rate	$^1/_{100}$

Table B.3: Sequential benchmark (variable *area-under-curve* value)

Container	Throughput [op/s]					
	0.45	0.55	0.65	0.75	0.85	0.95
Baseline						588
WST, aging = 0	1199	1432	1709	1997	2329	2890
WST, aging = 1	1103	1341	1541	1802	2183	2749
WST, aging = 2	987	1188	1375	1630	1931	2559
WST, aging = 4	858	983	1146	1321	1609	2325
WST, aging = 16	798	887	1028	1216	1560	2537
Hash table, Binary heap, aging = 0	1183	1416	1655	1904	2211	2733
Hash table, Binary heap, aging = 1	1185	1405	1651	1881	2220	2759
Hash table, Binary heap, aging = 2	1178	1403	1647	1897	2199	2784
Hash table, Binary heap, aging = 4	1188	1407	1651	1908	2204	2755
Hash table, Binary heap, aging = 16	1179	1391	1638	1845	2152	2675
Hash table, LRU	603	621	643	678	720	809
Hash table, LFU	576	591	604	624	651	700
Splay tree, Splay policy, canonical	658	698	747	801	896	1040
Splay tree, Splay policy, partial	616	669	767	781	799	861
Splay tree, LRU, canonical	550	566	582	595	631	687
Splay tree, LFU, canonical	567	577	590	608	628	675

Parameter	Value
Min/Max arg	25/35
Memory	64 KiB
Area-under-curve	*vary*
Mean changing rate	$^1/_{100}$

Table B.4: Sequential benchmark (variable mean changing rate)

Container	Throughput [op/s]			
	0	$^1/_{100}$	$^1/_{10}$	1
Baseline				588
WST, aging = 0	2461	2329	1161	330
WST, aging = 1	2279	2183	1169	344
WST, aging = 2	2052	1931	1136	349
WST, aging = 4	1699	1609	1054	350
WST, aging = 16	1721	1560	800	334
Hash table, Binary heap, aging = 0	2344	2211	1238	336
Hash table, Binary heap, aging = 1	2353	2220	1238	333
Hash table, Binary heap, aging = 2	2391	2199	1241	331
Hash table, Binary heap, aging = 4	2335	2204	1235	331
Hash table, Binary heap, aging = 16	2316	2152	1214	333
Hash table, LRU	771	720	479	386
Hash table, LFU	706	651	438	356
Splay tree, Splay policy, canonical	957	896	565	311
Splay tree, Splay policy, partial	823	799	533	398
Splay tree, LRU, canonical	679	631	419	350
Splay tree, LFU, canonical	659	628	418	340

Parameter	Value
Min/Max arg	25/35
Memory	64 KiB
Area-under-curve	0.85
Mean changing rate	*vary*

Table B.5: Parallel benchmark (variable memory size, static input distribution)

Container	Throughput [op/s]		
	4 KiB	16 KiB	64 KiB
Baseline			428
CNDC (1)	809	850	828
CNDC (2)	1251	1465	1576
CNDC (4)	2114	2505	3088
CNDC (8)	3217	4332	5023
CNDC (16)	5585	7823	9043
CNDC (24)	9062	11178	12920
Coarse lock + WST (1)	2262	2502	2421
Coarse lock + WST (2)	1051	1280	1190
Coarse lock + WST (4)	1276	1301	1232
Coarse lock + WST (8)	1340	1281	1166
Coarse lock + WST (16)	1407	1392	1266
Coarse lock + WST (24)	1439	1443	1236
Binning + WST (1)	1660	2302	2382
Binning + WST (2)	1786	2490	2864
Binning + WST (4)	1668	2720	3510
Binning + WST (8)	2633	3517	3789
Binning + WST (16)	3495	5144	5305
Binning + WST (24)	4295	5997	6367

Parameter	Value
Min/Max arg	25/35
Memory	*vary*
Area-under-curve	0.85
Mean changing rate	0

Table B.6: Parallel benchmark (variable memory size, time-varying input distribution)

Container	Throughput [op/s]		
	4 KiB	16 KiB	64 KiB
Baseline			428
CNDC (1)	729	650	734
CNDC (2)	1312	1404	1524
CNDC (4)	2148	2496	2984
CNDC (8)	3288	3968	5120
CNDC (16)	5744	6208	9040
CNDC (24)	9648	10824	12408
Coarse lock + WST (1)	1601	2155	2347
Coarse lock + WST (2)	848	1328	1230
Coarse lock + WST (4)	1264	1268	1200
Coarse lock + WST (8)	1320	1416	1272
Coarse lock + WST (16)	1489	1392	1248
Coarse lock + WST (24)	1560	1368	1272
Binning + WST (1)	1406	2052	2229
Binning + WST (2)	1540	2594	2794
Binning + WST (4)	1748	2852	3320
Binning + WST (8)	2728	3664	3792
Binning + WST (16)	4044	5232	5248
Binning + WST (24)	4296	5928	6384

Parameter	Value
Min/Max arg	25/35
Memory	*vary*
Area-under-curve	0.85
Mean changing rate	$\frac{1}{100 \times thread\ count}$

Appendix C

Acronyms

BST Binary Search Tree

T A binary tree

N A node in a binary tree or a hash table

K A key, used for the lookup in a container

F User-provided function that accepts K as an argument

R A result, that is calculated from K by F

Appendix D

Contents of enclosed CD

- /
  - Code ..... NUMDB project folder
    - 3rdparty ..... NUMDB additional libraries
      - function_traits
      - google_benchmark
      - gtest
      - murmurhash2functor
    - benchmark ..... NUMDB benchmark source files
    - include ..... NUMDB library header files
      - numdb
        - cndc
        - concurrent_adapters
        - hash_table
        - splay_tree
        - wst
    - lib ..... NUMDB library source files
    - test ..... NUMDB unit tests
  - Text
    - thesis.pdf ..... this thesis in the PDF format