A Simple Cache-Sensitive Skip List (SCSSL)

Yuchang Ke
aceking.ke@gmail.com
Your Institution Name Here
Your City, Your Country

Bill Lv
ideaalloc@gmail.com
Your Institution Name Here
Your City, Your Country

Weijie Zhang
zhangweijie@gmail.com
Your Institution Name Here
Your City, Your Country

Abstract

We present a cache-sensitive skip list (SCSSL) that is remarkably simple and easy to implement. It requires only minor modifications to the original skip list data structure to adapt it effectively to modern CPU architectures. This adaptation results in a performance improvement typically ranging from 1.2x to 1.6x, and on some systems, can reach 2x to 3x for insertion-heavy workloads. Unlike other optimization techniques, SCSSL does not rely on specialized hardware instructions like SIMD or intricate multi-threaded programming paradigms. Instead, its design inherently benefits from improved data locality due to the use of arrays at the data level, a feature that modern compilers can further leverage. This makes the proposed design broadly applicable, suitable not only for in-memory databases and key-value stores but also for resource-constrained embedded systems.

CCS Concepts

• Information systems → Data structures; **• Theory of computation** → *Storage management*.

Keywords

skip list, cache-sensitive, index structure, data structures, performance optimization

ACM Reference Format:
Yuchang Ke, Bill Lv, and Weijie Zhang. 2024. A Simple Cache-Sensitive Skip List (SCSSL). In *Proceedings of the Workshop on Simple Cache-Sensitive Skip Lists (SCSSL '24), Month DD, 2024, City, Country*. ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

In computer science, ordered data structures are fundamental for organizing and managing data efficiently. The performance of search, insertion, and deletion operations within these structures is crucial for a wide array of applications, as it directly impacts overall system performance. Numerous methods have been developed to achieve high performance in ordered data structures, such as AVL trees and Red-Black trees. However, these balanced tree structures are relatively complex to implement due to the necessity of rebalancing operations. This landscape changed when William Pugh introduced the skip list in 1990 [?]. The skip list, a probabilistic data structure, offered a simpler implementation alternative to traditional balanced trees while providing comparable performance characteristics.

Thanks to its simplicity and efficiency, the skip list has been widely adopted in various systems. Notable examples include databases like LevelDB [?], Redis [?], RocksDB [?], and HBase [?], as well as in-memory caches and the inverted index component of search engines like Lucene.

A standard skip list consists of multiple levels: a data level (level 0) that stores all data nodes in a sorted linked list, and multiple index levels above it. These index levels contain a subset of the nodes from lower levels and act as "express lanes" to expedite traversal to the desired data nodes. However, nodes in a skip list are typically allocated dynamically, leading to non-contiguous memory addresses. This fragmentation can degrade performance on modern CPUs due to frequent cache misses.

Although cache-sensitive skip list variants like the Cache-Sensitive Skip List (CSSL) [?] and ESL (Express Lane Skip List) [?] have been developed to better align with modern CPU architectures, they often introduce significant complexity. For instance, CSSL utilizes SIMD instructions, which may not be universally available or optimal across all platforms. ESL proposes merging index level nodes into arrays and employs sophisticated version-based locking protocols and lock-free operation logs for concurrency. We believe that the primary appeal of the skip list lies in its inherent simplicity. Overly complex cache-sensitive designs can diminish this advantage.

Our work is motivated by the observation that simple architectural changes can yield substantial performance gains by improving cache locality. We propose a modification that retains the core simplicity of the skip list while significantly enhancing its cache-friendliness.

2 Design Overview

In this paper, we present a Simple Cache-Sensitive Skip List (SCSSL). The core idea is to modify the nodes at the data level (level 0) to store a small, sorted array of elements (e.g., key-value pairs or just keys) instead of a single element. This approach leverages CPU cache locality for these small arrays, thereby enhancing performance. For example, Figure 1 illustrates a conceptual SCSSL where data nodes hold arrays of maximum size 2.

The design of SCSSL hinges on three key principles:

(1) **Array-based Data Nodes:** At the data level (level 0), each node contains a sorted array with a fixed maximum length. For integer keys, a maximum length that allows the node to fit well within CPU cache lines (e.g., up to 128 elements, depending on key/value sizes) is chosen.
(2) **Index Level Pivot:** For navigation through the index levels (levels 1 and above), only the first element (i.e., the smallest

key) of the array in a data-level node (or an index node pointing to such an array) is used as the pivot for comparison.

(3) **Insertion Handling:** The last element (i.e., the largest key) of an array in a data node is primarily considered during insertion operations, particularly when an array is full and a new key needs to be accommodated.

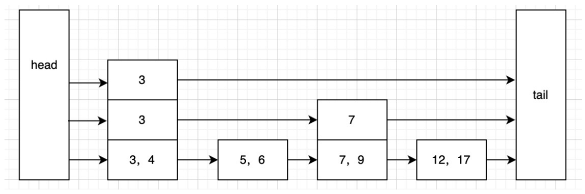

Figure 1: A conceptual view of the Simple Cache-Sensitive Skip List (SCSSL) where data nodes (level 0) store arrays of elements (e.g., max array size = 2 as depicted). Index nodes point to the start of these arrays.

Our key contributions include:

(1) Introducing arrays within data-level nodes, which maintains the asymptotic complexity of skip list operations while significantly improving CPU cache hit rates due to the contiguous storage of multiple elements.
(2) Reducing the effective height of the skip list and pointer overhead, as each data-level node now stores multiple elements, leading to fewer nodes overall for a given dataset size.
(3) Presenting modified insertion, search, and deletion algorithms tailored for these array-based data nodes, offering a practical and simple approach for enhancing skip list performance, particularly in database indexing and similar applications.

3 Algorithms

This section details the algorithms for search, insertion, and deletion in SCSSL. These operations are adapted from standard skip list algorithms to accommodate the array-based nodes at the data level.

In SCSSL, each node at the data level (level 0) stores its elements in a sorted array of a fixed maximum size. When a new node is created, its level is chosen randomly, similar to a standard skip list. A node promoted to level k has $k+1$ forward pointers (indexed 0 to k). The maximum number of levels in the skip list is a predefined constant, 'MAX$_P OSSIBLE_L EVEL$'$.Theskiplistemploysaheadernodetoinitiatesearches; itsforwardpointersareinitially$**NIL**$.Forsimplicityincomparisons, theh$

Figure 2 illustrates an example of an insertion operation in SCSSL.

3.1 Initialization

The initialization process is elementary. A header node is created with an array of 'MAX$_P OSSIBLE_L EVEL$+1'$forwardpointers, allinitializedto$**NIL**$.Tosimplifycomparisons, theheadernode'skeyarray(ifitweretohaveonei$
∞.

3.2 Search

The search algorithm (Algorithm 1) navigates the skip list from the highest level down to level 0. At each level 'i', it traverses

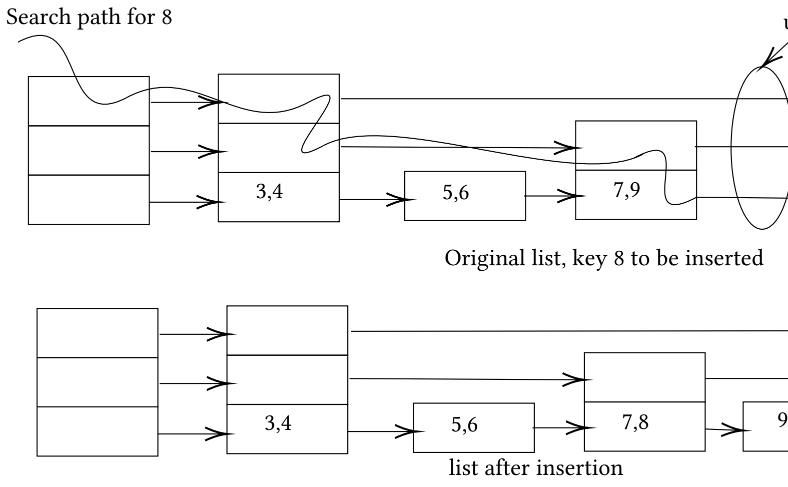

Figure 2: Insertion of key 8 into an SCSSL (max array size = 2). Key 8 displaces 9 from node (7,9). Key 9 is then inserted into a new node.

forward as long as the first key of the array in the next node ('current.forward[i].FirstKey()') is less than or equal to the search key 'k'. Once this traversal is complete, the 'current' node is the one whose own array might contain 'k' (specifically, 'current.FirstKey() <= k', and either 'current.forward[0]' is **NIL** or 'current.forward[0].FirstKey() > k'). The final step involves searching for 'k' within 'current''s array.

Algorithm 1: Search for Key k in SCSSL

```
Input: Search key $k$, current global maximum level
       'currentGlobalMaxLevel', header node 'header'
Output: Node containing key $k$ and its index in array, or
        (NIL, -1) if not found
1 Function Search($k$, $currentGlobalMaxLevel$, $header$):
2     $current \leftarrow header$;
3     for $i \leftarrow currentGlobalMaxLevel$ 0 do
4         while $current.forward[i] \neq$
            NIL $\land$ $current.forward[i].FirstKey() \leq k$ do
5             $current \leftarrow current.forward[i]$;
6     return $current$.search_in_array($k$);
      // Returns (node, index) or (NIL, -1)
```

3.3 Insertion

The insertion algorithm (Algorithm 2) first searches for the appropriate position for the new key 'k', maintaining an 'update' array to store pointers to the predecessor nodes at each level. The core logic then addresses several cases based on whether the target data node ('target$_n ode$', $whichis$'$update[0]$'$)canaccommodate$'k' :

Current node is header: If 'target$_n ode$'$istheheader, anewnodeisusuallycre$

Target node full, ‘k‘ fits within existing range (CASE 1 from paper): If ‘target$_node‘'sarrayisfullbut‘k‘issmallerthanitslargestelement(‘target_node.array_last_element()‘)and‘k‘isnotsmallerthanitsfirstele$

Target node not full (CASE 2 from paper): If ‘target$_node‘'sarrayisnotfulland‘k‘belongsthere(i.e.,‘k>= target_node.FirstKey()‘),‘k‘isinserteddirectlyinto‘target_node‘'sarray.Theoperationthencompletes.$

**Key ‘k‘ belongs after ‘target$_n$$ode‘(CASE3frompaper): If‘k‘isgreaterthanallelementsin‘target_node‘'sarray(‘k > target_node.array_last_element()‘),insertionisattemptedin‘target_node.forward[0]‘.If‘targ$

New node creation (Other CASES): If none of the above conditions lead to an in-place insertion, a new node is created for ‘k‘. Its level is randomly determined. If this new level exceeds the list's ‘currentGlobalMaxLevel‘, the ‘update‘ array and ‘currentGlobalMaxLevel‘ are adjusted. Pointers are then updated to link the new node.

3.4 Deletion

The deletion algorithm (Algorithm 3) also starts by finding the key ‘k‘. The traversal logic advances ‘current‘ as long as the *entire array* in ‘current.forward[i]‘ contains keys strictly smaller than ‘k‘ (i.e., ‘current.forward[i].array$_last_element() < k‘).Thisensuresthat‘update[0].forward[0]‘becomesthefirstnodewhosearraycouldpotentiallycontain‘k‘.If‘k‘isfoundanddeletedfromthisnode'sa$

If the array becomes empty, the node itself is removed from the skip list by adjusting the forward pointers of its predecessors.
If removing a node causes some highest levels to become empty, ‘currentGlobalMaxLevel‘ is decremented.

4 Experimental Evaluation

4.1 Experimental Environment

The experiments were conducted in the following environments:

- **Hardware 1 (MacBook):** Apple M2 CPU, 16 GB RAM.
- **Hardware 2 (Server):** Intel(R) Core(TM) i7-6700 CPU @ 3.40GHz, 24 GB DDR4 2133 MHz RAM. This server was used not only for performance testing but also specifically for analyzing CPU cache miss rates using appropriate profiling tools (e.g., ‘perf‘).
- **Compiler:** The algorithms were implemented in C++. The test programs were compiled using g++ (version specific to environment, e.g., 9.0 or higher) with the ‘-O3‘ optimization flag.
- **Benchmark Configuration:** Insertion operations in a skip list are prone to CPU cache misses due to dynamic memory allocations and pointer chasing. Therefore, our benchmarks primarily focus on insertion performance using randomly generated integer keys. The number of elements ranged from 100 to 300,000. Search performance was also evaluated. However, we do not present detailed search results as the performance difference between SCSSL and the

Algorithm 2: Insert Key k into SCSSL

```
Input: Key k, current global maximum level
       ‘currentGlobalMaxLevel‘, header node ‘header‘,
       constant ‘MAX_POSSIBLE_LEVEL‘
Output: Potentially updated ‘currentGlobalMaxLevel‘
Function Insert(k, currentGlobalMaxLevel, header,
  MAX_POSSIBLE_LEVEL):
    update[MAX_POSSIBLE_LEVEL + 1] ;   // Array to
      store predecessors
    current ← header;
    for i ← currentGlobalMaxLevel 0 do
        while current.forward[i] ≠
          NIL ∧ current.forward[i].FirstKey() ≤ k do
            current ← current.forward[i];
        update[i] ← current;
    target_node ← update[0];
    if target_node ≠ header ∧ target_node.is_full() ∧ k <
      target_node.array_last_element() ∧ k >=
      target_node.FirstKey() then
        // CASE 1
        replaced_key ← target_node.array.pop_last();
        target_node.insert_into_array(k);
        k ← replaced_key;
    if
      target_node ≠ header ∧ ¬target_node.is_full() ∧ k >=
      target_node.FirstKey() then
        // CASE 2
        target_node.insert_into_array(k);
        return currentGlobalMaxLevel;
        // Insertion complete
    current ← target_node.forward[0];
    if current ≠ NIL ∧ ¬current.is_full() ∧ k >=
      current.FirstKey() then
        // CASE 3
        current.insert_into_array(k);
        return currentGlobalMaxLevel;
        // Insertion complete
    else
        // OTHER CASES: Create a new node for k
        new_node_level ← random_level();
        if new_node_level > currentGlobalMaxLevel then
            for i ← currentGlobalMaxLevel + 1 to
              new_node_level do
                update[i] ← header;
            currentGlobalMaxLevel ← new_node_level;
        new_node ← Node([k], new_node_level);
        for i ← 0 to new_node_level do
            new_node.forward[i] ←
              update[i].forward[i];
            update[i].forward[i] ← new_node;
    return currentGlobalMaxLevel;
```

Algorithm 3: Delete Key k from SCSSL

```
Input: Key k, current global maximum level
       ‘currentGlobalMaxLevel‘, header node ‘header‘,
       constant ‘MAX_P OSSIBLE_L EVEL‘
Output: (Boolean indicating success, potentially updated
        ‘currentGlobalMaxLevel‘)
Function Delete(k, currentGlobalMaxLevel, header,
  MAX_POSSIBLE_LEVEL):
    update[MAX_POSSIBLE_LEVEL + 1];
    current ← header;
    for i ← currentGlobalMaxLevel 0 do
        while current.forward[i] ≠ NIL ∧
          current.forward[i].array_last_element() < k do
            current ← current.forward[i];
        update[i] ← current;
    node_to_check ← current.forward[0];
    if node_to_check = NIL then
        return (False, currentGlobalMaxLevel);
    if k < node_to_check.FirstKey() then
        return (False, currentGlobalMaxLevel);
    deleted_from_arr ←
      node_to_check.delete_from_array(k);
    if ¬deleted_from_arr then
        return (False, currentGlobalMaxLevel);
    if node_to_check.is_array_empty() then
        for i ← 0 to node_to_check.level do
            if update[i].forward[i] ≠ node_to_check then
                ;
            update[i].forward[i] ←
              node_to_check.forward[i];
        while currentGlobalMaxLevel > 0 ∧
          header.forward[currentGlobalMaxLevel] = NIL
          do
            currentGlobalMaxLevel ←
              currentGlobalMaxLevel − 1;
    return (True, currentGlobalMaxLevel);
```

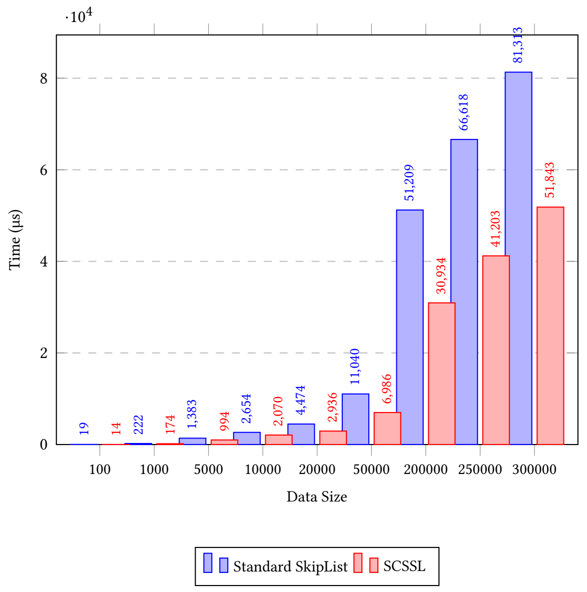

Figure 3: Insertion Time Comparison: Standard Skip List vs. SCSSL on macOS (Apple M2). Lower is better.

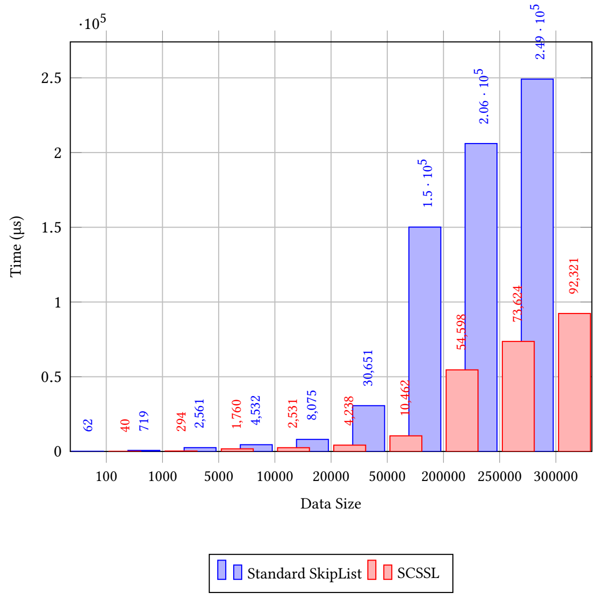

Figure 4: Insertion Time Comparison: Standard Skip List vs. SCSSL on Server (Intel i7-6700). Lower is better.

standard skip list was found to be negligible for typical search operations. This is likely because the dominant factor in search is the traversal of levels, which has a similar path length, and the final array scan in SCSSL is very fast for small arrays. Both the standard skip list and SCSSL were configured with a maximum level (‘MAX$_P$$OSSIBLE_L$$EVEL$‘)$of16.ForSCSSL, themaximumarraysizewithinadatanodewassettoavalueempiricallyfoundtobeeffective(e.$

4.2 Results and Discussion

On the MacBook (Apple M2 processor, Figure 3), SCSSL demonstrates a speedup of 1.28x-1.39x for small datasets (<1,000 elements) during insertion. For medium-scale datasets (>20,000 elements), the speedup increases to a range of 1.55x-1.65x. This trend indicates that the cache-friendly benefits of SCSSL become more pronounced as the dataset size grows, leading to more memory accesses.

On the server system (Intel i7-6700, Figure 4), SCSSL achieves even more significant speedups for insertions, reaching up to nearly 3x compared to the standard skip list for larger datasets (e.g., 300,000 elements). This enhanced improvement on the server platform might be attributed to differences in cache architecture, memory subsystem performance, or the relative cost of cache misses compared to the M2 architecture.

Table 1: Cache Performance Comparison (Insertions, 300,000 elements) on Server (Intel i7-6700)

Test Case	Cache References	Cache Misses	Cache Miss Rate
SCSSL	51,567,218	459,636	0.89%
Standard Skip List	142,249,678	47,251,323	33.22%

Table 1 presents the cache performance data for inserting 300,000 random elements on the server. The standard skip list exhibits a high L1 data cache miss rate of 33.22%. In contrast, SCSSL drastically reduces this rate to a mere 0.89%. This substantial reduction in cache misses directly corroborates our hypothesis that embedding arrays within data nodes effectively improves data locality and cache utilization, which translates into the observed speedups. The significantly lower number of cache references for SCSSL also suggests more efficient data access patterns.

5 Conclusion

We have presented SCSSL, a novel cache-sensitive skip list variant that introduces a simple yet highly effective modification: replacing single elements in data-level nodes with small, sorted arrays. This architectural change allows SCSSL to enhance CPU cache performance by improving data locality, while crucially maintaining the algorithmic simplicity that is a hallmark of skip lists.

Experimental results demonstrate significant performance improvements for insertion operations, particularly on systems where cache performance is a critical factor. The observed speedups of 1.2x-3x, coupled with a dramatic reduction in cache miss rates (e.g., from 33.22% to 0.89% in our server tests), validate the efficacy of this approach. SCSSL offers a practical and easily implementable method to boost skip list performance in modern computing environments without resorting to complex algorithmic changes or specialized hardware features. Its simplicity makes it an attractive option for a wide range of applications, from embedded systems to large-scale in-memory data stores.

Future work could explore optimal array sizes for different hardware platforms and key/value types, as well as investigate the application of SCSSL principles in concurrent skip list implementations.