IMPLEMENTATION OF SKIPLIST

TECHNICAL CODETHAN REPORT
SUBMITTED TO

RAMAIAH INSTITUTE OF TECHNOLOGY
(Autonomous Institute, Affiliated to VTU)

Bangalore – 560054

SUBMITTED BY

LAYA ARUN	**1MS21CS067**
MANIK ANIMESH AJIT	**1MS21CS071**
NATASHA SURESH	**1MS21CS081**
PRANAY SHARMA	**1MS21CS097**

As part of the Course **Data Structures– CS33**

SUPERVISED BY
Faculty
Nandini S B

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

RAMAIAH INSTITUTE OF TECHNOLOGY

Dec-2022

Department of Computer Science and Engineering

Ramaiah Institute of Technology
(Autonomous Institute, Affiliated to VTU)

Bangalore – 54

CERTIFICATE

This is to certify that

LAYA ARUN (1MS21CS067),

MANIK ANIMESH AJIT (1MS21CS071)

NATASHA SURESH (1MS21CS081)

PRANAY SHARMA (1MS21CS097)

have completed the **"IMPLEMENTATION OF SKIPLIST"** as part of Technical Codethan. We declare that the entire content embodied in this B.E. 3rd Semester report contents are not copied.

Submitted by:

LAYA ARUN (1MS21CS067)
MANIK ANIMESH AJIT (1MS21CS071)
NATASHA SURESH (1MS21CS081)
PRANAY SHARMA (1MS21CS097)

Guided by:

Nandini S B
(Dept of CSE, RIT)

Department of Computer Science and Engineering

Ramaiah Institute of Technology
(Autonomous Institute, Affiliated to VTU)

Bangalore – 54

Evaluation Sheet

Sl. No.	USN	Name	Research Content understanding and Coding (10)	Demo & Report submission (10)	Total Marks (20)
1.	1MS21CS067	LAYA ARUN			
2.	1MS21CS071	MANIK ANIMESH AJIT			
3.	1MS21CS081	NATASHA SURESH			
4.	1MS21CS097	PRANAY SHARMA			

Evaluated By

Name: Nandini S B
Designation: Assistant Professor
Department: Computer Science & Engineering, RIT
Signature:

HOD, CSE

Table of Contents

1) <u>ABSTRACT</u>

Recent advances in memory architectures have provoked renewed interest in neardata-processing (NDP) as way to alleviate the “memory wall” problem. An NDP architecture places logic circuits, such as simple processors, in close proximity to memory. Effective use of NDP architectures requires rethinking data structures and their algorithms. Here, we provide an empirical evaluation of several NDP-aware algorithms for general-purpose concurrent data structures such as linked-lists, skiplists, and FIFO queues. The empirical analysis reveals that the potential benefits of NDP-based concurrent data structures are less than what had been expected in earlier studies. In turn, we introduce lightweight NDP hardware modifications, inspired by initial observations on data access patterns and underlying DRAM activity. Even the minimal changes to hardware significantly improve the performance and energy consumption of NDP-based concurrent data structures, and in many cases, the resulting data structures outperform state-of-the-art concurrent data structures.

2) INTRODUCTION

The increasing discrepancy between processor speeds and memory access speeds (often referred to as memory wall) causes memory access to be the principal performance bottleneck in many of today's data intensive applications. Until recently, most architectures have relied on multi-level caches to reduce data access latency. However, caches have become less and less effective over time. As data intensive applications increasingly use data sets much larger than cache sizes and exhibit irregular and unpredictable memory access patterns, it is hard to simply rely on caches to improve application performance. Moreover, frequent data movement between host processors and memory causes high energy consumption, a growing concern with data intensive applications.

We focus on software libraries and architectural support for general purpose concurrent data structures with near-data-processing architectures. These data structures are used in many applications, and adapting them to NDP architectures is a key step toward making these architectures useful. In conventional architectures, "pointer-chasing" data structures with poor cache locality and high-contention concurrent data structures are often bottlenecks, while neardata-processing architectures have the promise to alleviate or even eliminate these problems.

In this paper, we implement and test those data structure algorithms on a full system NDP architecture framework with realistic hardware constraints.

Liu et al observed that naïve implementations of data structures on near-data-processing architecture will serialize data structure operations and will be outperformed by highly concurrent state-of-the-art data structures on conventional architectures. As an algorithmic solution, they proposed using flat-combining techniques to add concurrency to NDP-based data structures.

Through this more realistic and detailed analysis, we find that Liu et al's work is incomplete. Although the results show that flat-combining does indeed enhance concurrency in NDP-based data structures, the resulting performance falls short compared to what was expected in the theoretical analysis. We identify that the theoretical analysis had two major pitfalls: (1) it ignored the cache impacts in host-based concurrent data structure performance, and (2) it had overly optimistic assumptions on near- data-processing memory access latencies. These pitfalls led to the overestimated relative performance of NDP-based concurrent data structures.

To address these shortcomings, we show that lightweight changes to hardware can reduce the performance gap while using the same algorithm. The hardware changes were inspired by observations on data structures' access patterns and underlying DRAM activity.

3) LITERATURE SURVEY

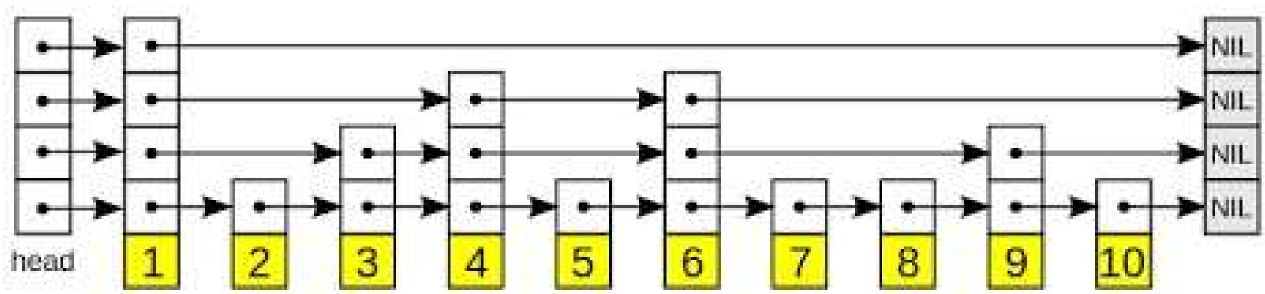
This paper makes the following contributions:

- We define a generic near-data-processing (NDP) architecture that is wellsuited for concurrent data structures.
- We implement actual software kernels of the NDP-based concurrent data structures on a cycle-accurate full-system NDP architecture framework, yielding a more realistic and detailed analysis in terms of performance, energy, and power.
- Using our architecture framework, we identify the shortcomings of prior theoretical analyses that led to overestimated relative performance of NDP-based concurrent data structures.
- The findings from this evaluation suggest lightweight adjustments to hardware design. We show that minimal hardware changes can significantly improve the performance and energy consumption of NDPbased conurrent data structures.

The One that we understood and Implement is SKIPLIST.

SKIPLIST

The skiplist is another type of pointer-chasing data structure. Skiplist nodes hold multiple levels of pointers, and the pointer at each level points to the following node at that same level. Each node is assigned a random maximum level, taken from a particular distribution, to provide balanced tree-like characteristics. The nodes in the skiplist are also ordered in ascending order of integer keys. The NDPbased skiplist is optimized by partitioning the skiplist across multiple NDP vaults based on pre-defined disjoint ranges of keys, as shown in Figure 2b. We assume that the host processors are provided with the range of keys belonging to each NDP vault. Host processors send operation requests to the appropriate NDP core, based on the requested operation key. Each NDP core acts as the combiner for its designated partition, which takes care of synchronization. Partitioning also enables multiple NDP cores to execute operations in parallel.



4) ABSTRACT DATA TYPE

ADT in SKIPLIST are:- typedef

```
struct node {
   int key;
   int value;
   struct node **above; }
node;
```

The node of ADT is the fundamental unit of data. In this structure, each node has a key, value and a pointer. The pointer points to the next node in the list. This will make it easier for us to traverse through the list by following the pointers.

```
typedef struct skiplist {
   int level;
int size;
   struct node *lead;
} skiplist;
```

This structure consists of level in the skiplist and size of the skiplist at particular level and the pointer of each node at different levels.

BASIC OPERATIONS OF SKIPLIST:-

There are the following types of operations in the skip list.

- **Insertion operation:** It is used to add a new node to a particular location in a specific situation.

```
int skiplist_insert(skiplist *list, int key, int value) {
node *update[SKIPLIST_MAX_LEVEL + 1];
    node *x = list->lead;
    int i, level;     for (i = list->level; i
>= 1; i--) {         while (x->above[i]-
>key < key)
            x = x->above[i];
update[i] = x;
    }
    x = x->above[1];

    if (key == x->key) {
x->value = value;         return
0;
        // element already exists
    }      else {         level =
rand_level();         if (level
> list->level) {
            for (i = list->level + 1; i <= level; i++) {
update[i] = list->lead;
            }
            list->level = level;
        }

        x = (node *) malloc(sizeof(node));
x->key = key;         x->value = value;
        x->above = (node **) malloc(sizeof(node*) * (level + 1));
        for (i = 1; i <= level; i++) {             x-
>above[i] = update[i]->above[i];
update[i]->above[i] = x;
        }
    }

    return 0;
}
```

- **Deletion operation:** It is used to delete a node in a specific situation

```
int skiplist_delete(skiplist *list, int key) {
int i;
```

```
   node *update[SKIPLIST_MAX_LEVEL + 1];
   node *x = list->lead;     for (i =
list->level; i >= 1; i--) {         while
(x->above[i]->key < key)
          x = x->above[i];
update[i] = x;
   }

   x = x->above[1];     if (x->key ==
key) {         for (i = 1; i <= list-
>level; i++) {             if (update[i]-
>above[i] != x)
             break;
          update[i]->above[1] = x->above[i];
       }
       skiplist_node_free(x);

       while (list->level > 1 && list->lead->above[list->level]
             == list->lead)
list->level--;         return 0;
   }
return 1;
   //for attempt to remove non - existent node
}
```

- **Search Operation:** The search operation is used to search a particular node in a skip list

```
node *skiplist_search(skiplist *list, int key) {
node *x = list->lead;
   int i;
   for (i = list->level; i >= 1; i--) {
while (x->above[i]->key < key)
          x = x->above[i];
   }
   if (x->above[1]->key == key) {
      return x->above[1];
   } else {
return NULL;
   }   return NULL; }
```

5) IMPLEMANTATION

```
#include <stdlib.h>
#include <stdio.h>
#include <stdbool.h>
#include <limits.h>
#include <time.h>

#define SKIPLIST_MAX_LEVEL 6 const
int negative_inf = INT_MIN;
const int positive_inf = INT_MAX;

#ifdef laya1234
   freopen("input.txt", "r", stdin);
freopen("output.txt", "w", stdout);
freopen("misc.txt", "w", stderr);
#endif

typedef struct node {
   int key;
   int value;      struct
node **above;
} node;

typedef struct skiplist {
   int level;    int
size;    struct node
*lead;
} skiplist;

skiplist *skiplist_init(skiplist *list) {
int i;
   node *header = (node *) malloc(sizeof(struct node));
   list->lead = header;    header->key = positive_inf;
header->above = (node **) malloc(           sizeof(node*)
```

```
* (SKIPLIST_MAX_LEVEL + 1));     for (i = 0; i <=
SKIPLIST_MAX_LEVEL; i++) {
        header->above[i] = list->lead;
    }

    list->level = 1;
    list->size = 0;

    return list;
}

static int rand_level() {
int level = 1;
srand(time(NULL));
    while (rand()%2 && level < SKIPLIST_MAX_LEVEL)
level++;

    //while(rand() < RAND_MAX / 2 && level<
SKIPLIST_MAX_LEVEL) level++;

    return level;
}

int skiplist_insert(skiplist *list, int key, int value) {
node *update[SKIPLIST_MAX_LEVEL + 1];
    node *x = list->lead;
    int i, level;     for (i = list->level; i
>= 1; i--) {        while (x->above[i]-
>key < key)
            x = x->above[i];
update[i] = x;
    }
    x = x->above[1];

    if (key == x->key) {
x->value = value;        return
0;
        // element already exists
```

```
    }      else {          level = 
rand_level();         if (level 
> list->level) {
            for (i = list->level + 1; i <= level; i++) {
                update[i] = list->lead;
            }
            list->level = level;
        }

        x = (node *) malloc(sizeof(node));
x->key = key;          x->value = value;
        x->above = (node **) malloc(sizeof(node*) * (level + 1));
        for (i = 1; i <= level; i++) {              x-
>above[i] = update[i]->above[i];
update[i]->above[i] = x;
        }
    }

    return 0;
}

node *skiplist_search(skiplist *list, int key) {
node *x = list->lead;
    int i;
    for (i = list->level; i >= 1; i--) {
while (x->above[i]->key < key)
            x = x->above[i];
    }
    if (x->above[1]->key == key) {
return x->above[1];
    } else {
return NULL;
    }
    return NULL;
}
```

```
static void skiplist_node_free(node *x) {
if (x) {        free(x->above);
free(x);
    }
}

int skiplist_delete(skiplist *list, int key) {
int i;
    node *update[SKIPLIST_MAX_LEVEL + 1];
    node *x = list->lead;     for (i =
list->level; i >= 1; i--) {         while
(x->above[i]->key < key)
            x = x->above[i];
update[i] = x;
    }

    x = x->above[1];     if (x->key ==
key) {         for (i = 1; i <= list-
>level; i++) {            if (update[i]-
>above[i] != x)
                break;
            update[i]->above[1] = x->above[i];
        }
        skiplist_node_free(x);

        while (list->level > 1 && list->lead->above[list->level]
                == list->lead)
list->level--;        return 0;
    }
return 1;
    //for attempt to remove non - existent node
}

static void skiplist_dump(skiplist *list) {
printf("-INF->");     node *x = list-
>lead;
    while (x && x->above[1] != list->lead) {
        printf("%d[%d]->", x->above[1]->key, x->above[1]->value);
x = x->above[1];
```

```
    }
    printf("INF\n");
}

int main() {
    int arr[] = { 3, 6, 9, 2, 11, 72, 1, 4 };
    int i;
skiplist list;
    skiplist_init(&list);

    printf("Insert:--------------------\n");     for (i =
0; i < sizeof(arr) / sizeof(arr[0]); i++) {
skiplist_insert(&list, arr[i], arr[i]);
    }
    skiplist_dump(&list);


    printf("Search:--------------------\n");
    int keys[] = { 3, 4, 7, 10, 111, 72 };

    for (i = 0; i < sizeof(keys) / sizeof(keys[0]); i++) {
        node *x = skiplist_search(&list, keys[i]);
if (x) {
            printf("key = %d, value = %d\n", keys[i], x->value);
        } else {
            printf("key = %d, not found\n", keys[i]);
        }
    }

    printf("Delete:--------------------\n");

    skiplist_delete(&list, 3);
skiplist_delete(&list, 9);     skiplist_delete(&list,
72);     skiplist_delete(&list, 1);
    skiplist_dump(&list);

    return 0;
}
```

6) <u>RESULTS</u>

```
❯ gcc check.c
❯ ./a.out
Insert:-------------------
-INF->1[1]->2[2]->3[3]->4[4]->6[6]->9[9]->11[11]->72[72]->INF
Search:-------------------
key = 3, value = 3
key = 4, value = 4
key = 7, not found
key = 10, not found
key = 111, not found
key = 72, value = 72
Delete:-------------------
-INF->2[2]->4[4]->6[6]->11[11]->INF
```

7) CONCLUSION

- If you want to insert a new node in the skip list, then it will insert the node very fast because there are no rotations in the skip list.
- The skip list is simple to implement as compared to the hash table and the binary search tree.
- It is very simple to find a node in the list because it stores the nodes in sorted form.
- It requires more memory than the balanced tree.
- Reverse searching is not allowed.
- The skip list searches the node much slower than the linked list.
- It is used in distributed applications, and it represents the pointers and system in the distributed applications.
- It is used to implement a dynamic elastic concurrent queue with low lock contention.
- It is also used with the QMap template class.
- The indexing of the skip list is used in running median problems.
- The skip list is used for the delta-encoding posting in the Lucene search.

8) REFERENCES

- **Research paper**- Concurrent Data Structures with Near-Data-Processing: an Architecture-Aware Implementation .

[1] Paula Aguilera, Dong Ping Zhang, Nam Sung Kim, and Nuwan Jayasena. 2016. FineGrained Task Migration for Graph Algorithms using Processing in Memory. In Parallel and Distributed Processing Symposium Workshops, 2016 IEEE International. IEEE, 489– 498.

[2] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. 2015. A scalable processing-in-memory accelerator for parallel graph processing. In Computer Architecture (ISCA), 2015 ACM/IEEE 42nd Annual International Symposium on. IEEE, 105–117.

[3] Junwhan Ahn, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. 2015. PIM-enabled instructions: A low-overhead, locality-aware processing-in-memory architecture. In Computer Architecture (ISCA), 2015 ACM/IEEE 42nd Annual International Symposium on. IEEE, 336–348.

[4] JEDEC Solid State Technology Association. 2013. High Bandwidth Memory (HBM) DRAM. Standard JESD235 (2013).

[5] Erfan Azarkhish, Davide Rossi, Igor Loi, and Luca Benini. 2016. Design and evaluation of a processing-in-memory architecture for the smart memory cube. In International Conference on Architecture of Computing Systems. Springer, 19–31.

[6] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi,

Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The Gem5 simulator. ACM SIGARCH Computer Architecture News 39, 2 (2011), 1–7.

[7] Amirali Boroumand, Saugata Ghose, Youngsok Kim, Rachata Ausavarungnirun, Eric Shiu, Rahul Thakur, Daehyun Kim, Aki Kuusela, Allan Knies, Parthasarathy Ranganathan, and Onur Mutlu. 2018. Google Workloads for Consumer Devices: Mitigating Data Movement Bottlenecks. In Proceedings of the TwentyThird International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '18). ACM, New York, NY, USA, 316–331. https://doi.org/10.1145/3173162.3173177 .

[8] Amirali Boroumand, Saugata Ghose, Brandon Lucia, Kevin Hsieh, Krishna Malladi, Hongzhong Zheng, and Onur Mutlu. 2017. LazyPIM: An efficient cache coherence mechanism for processing-in-memory. IEEE Computer Architecture Letters (2017).

[9] Karthik Chandrasekar, Benny Akesson, and Kees Goossens. 2011. Improved power modeling of DDR SDRAMs. In Digital System Design (DSD), 2011 14th Euromicro Conference on. IEEE, 99–108.

[10] ARM Cortex. 2013. A15 MPCore processor technical reference manual. ARM Limited, Jun 24 (2013), 12.

[11] Palash Das, Shivam Lakhotia, Prabodh Shetty, and Hemangee K Kapoor. 2018. Towards Near Data Processing of Convolutional Neural Networks. In VLSI Design and 2018 17th International Conference on Embedded Systems (VLSID), 2018 31st International Conference on. IEEE, 380–385.

[12] Fernando A. Endo, Damien Courousse, and Henri-Pierre Charles. 2015. Microarchitectural Simulation of Embedded Core Heterogeneity with Gem5 and McPAT. In Proceedings of the 2015 Workshop on Rapid Simulation and Performance Evaluation:

Methods and Tools (RAPIDO ’15). ACM, New York, NY, USA, Article 7, 6 pages. https://doi.org/10.1145/2693433.2693440.

[13] Mikhail Fomitchev and Eric Ruppert. 2004. Lock-free Linked Lists and Skip Lists. In Proceedings of the Twenty-third Annual ACM Symposium on Principles of Distributed

Computing (PODC ’04). ACM, New York, NY, USA, 50–59. https://doi.org/10.1145/1011767.1011776 .

[14] Daichi Fujiki, Scott Mahlke, and Reetuparna Das. 2018. In-Memory Data Parallel Processor. In Proceedings of the Twenty-Third International Conference on Architectural

Support for Programming Languages and Operating Systems (ASPLOS ’18). ACM, New York, NY, USA, 1–14. https://doi.org/10.1145/3173162.3173171.

[15] M. Gao, G. Ayers, and C. Kozyrakis. 2015. Practical Near-Data Processing for InMemory Analytics Frameworks. In 2015 International Conference on Parallel

Architecture and Compilation (PACT). 113–124. https://doi.org/10.1109/PACT. 2015.22