1 Critical Functions: Add, Find, Delete

Operations that application threads call. Algorithm 1 defines a common procedure across these functions: traversing from the numa local Index Layer into the Data Layer. The Index Layer represents a skip list, acting as an attached, relaxed "search layer" to the Data Layer which represents a standard linked list. Our goals: maximize cache hits of numa local zones and provide a write-back, asynchronous update policy in the Index Layer while maintaining correctness in the Data Layer.

Algorithm 1 Traverse Into the Data Layer, Starting At the Index Layer

```
procedure GetElement($Sentinel, val$)          ▷ Sentinel - index layer node, val - targeted integer
    $previous \leftarrow sentinel$
    for $i \leftarrow previous.topLevel - 1$ downto 0 do
        $current \leftarrow previous.next[i]$
        while $current.val < val$ do
            $previous \leftarrow current$
            $current \leftarrow current.next[i]$
    return $previous.dataLayer$
```

Algorithm 2 Find

```
function Find($IndexLayer, val$)          ▷ IndexLayer - numa local search layer, val - targeted integer
    $current \leftarrow GetElement(IndexLayer.sentinel, val)$
    while $current.val < val$ do
        $current \leftarrow current.next$
    return $(current.val == val)$ & $(current.markedToDelete$ is $false)$
```

Algorithm 3 Add

```
global variable $numberNumaZones$, stores the number of numa zones on the machine
function Add($IndexLayer, val$)
    while $true$ do
        $previous \leftarrow getElement(IndexLayer.sentinel, val)$
        $current \leftarrow previous.next$
        while $current.val < val$ do
            $previous \leftarrow current$
            $current \leftarrow current.next$
        $lock(previous.lock)$
        $lock(current.lock)$
        if $validateLink(previous, current)$ is true then
            if $current.val == val$ then
                $unlock(previous)$
                $unlock(current)$
                return $false$
            else
                $insertion \leftarrow constructNode(val, numberNumaZones)$          ▷ Automatically sets fresh to true
                $insertion.next \leftarrow current$
                $previous.next \leftarrow insertion$
                $unlock(previous)$
                $unlock(current)$
                return $true$
        $unlock(previous)$
        $unlock(current)$
```

Algorithm 4 Remove Element

```
function REMOVE($IndexLayer, val$)        ▷ Where IndexLayer - numa local search layer, val - integer
    $previous \leftarrow getElement(IndexLayer.sentinel, val)$
    $current \leftarrow previous.next$
    while $current.val < val$ do
        $previous \leftarrow current$
        $current \leftarrow current.next$
    if $current.val \neq val$ OR $current.markedToDelete$ is $true$ then
        return $false$
    else if $CAS(current.markedToDelete, 0, 1) == 0$ then
        $current.fresh \leftarrow 1$
        return $true$
    else
        return $false$
```

2 Data Layer Utility Functions

Algorithm 5 Background Cleanup Thread

```
function BACKGROUNDREMOVAL($DataLayer$)        ▷ Where DataLayer - linked list of data layer nodes
    $sentinel \leftarrow DataLayer.sentinel$
    while $finished$ is $false$ do
        $previous \leftarrow sentinel$
        $current \leftarrow sentinel.next$
        while $current.next \neq NULL$ do
            if $current.fresh$ is $true$ then
                $current.fresh \leftarrow false$
                if $current.markedToDelete$ is $true$ then
                    $dispatchSignal(current, REMOVE)$
                else
                    $dispatchSignal(current, INSERT)$
            else if $current.markedToDelete$ is $true$ and $current.references == 0$ then
                $lock(previous.lock)$
                $lock(current.lock)$
                $valid \leftarrow validateLink(previous, current)$
                if $valid$ is $true$ then
                    $previous.next \leftarrow current.next$
                $unlock(previous.lock)$
                $unlock(current.lock)$
                if $valid$ is $true$ then
                    $current \leftarrow current.next$
                    $continue$
            $previous \leftarrow current$
            $current \leftarrow current.next$
```

Algorithm 6 Validate Links

```
function VALIDATELINK($previous, next$)        ▷ previous, next - data layer nodes
    return $previous.next == current$
```

Algorithm 7 Send a Job to Index Layers' Single Producer Single Consumer Queues

```
global variable $numberNumaZones$, stores the number of numa zones on the machine
global variable $indexLayers$, stores the index layer of each numa zone
function DISPATCHSIGNAL($node, operation$)        ▷ node - data layer node, operation - type of job to perform
    for $i \leftarrow 0$ to $numberNumaZones$ do
        $push(indexLayers[i].updateQueue, node, operation)$
```

Algorithm 8 Size

```
function SIZE($sentinel$)    ▷ sentinel - data layer sentinel node
    $runner \leftarrow setninel$
    $size \leftarrow -1$
    while $runner.next \neq NULL$ do
        if $runner.markedToDelete$ is $false$ then
            $size \leftarrow size + 1$
        $runner \leftarrow runner.next$
    return $size$
```

3 Index Layer Functions

Algorithm 9 Add Element to Index Layer

```
function ADD($Sentinel, dlNode, zone$)▷ Where Sentinel - index layer sentinel node, dlNode - data layer node to be
replicated, zone - integer of local numa zone
    $predecessors[sentintel.topLevel]$ is an array of index layer nodes
    $successors[sentintel.topLevel]$ is an array of index layer nodes
    $previous \leftarrow sentinel$
    $current \leftarrow NULL$
    $val \leftarrow dlNode$
    for $i \leftarrow previous.topLevel - 1$ downto 0 do
        $current \leftarrow previous.next[i]$
        while $current.val < val$ do
            $previous \leftarrow current$
            $current \leftarrow current.next[i]$
        $predecessors[i] \leftarrow previous$
        $successors[i] \leftarrow current$
    $candidate \leftarrow current$
    if $candidate.val \neq val$ then
        $topLevel \leftarrow getRandomLevel(sentinel.topLevel)$
        $insertion \leftarrow constructIndexNode(val, topLevel, dlNode, zone)$
        for $i \leftarrow 0$ to $topLevel - 1$ do
            $insertion.next[i] \leftarrow succesors[i]$
            $predecessors[i].next[i] \leftarrow insertion$
        return $true$
    return $false$
```

Algorithm 10 Remove Element from Index Layer

```
function REMOVE($Sentinel, val, zone$)    ▷ Where Sentinel - index layer sentinel node, val - targeted value, zone -
integer of local numa zone
    $predecessors[sentintel.topLevel]$ is an array of index layer nodes
    $successors[sentintel.topLevel]$ is an array of index layer nodes
    $previous \leftarrow sentinel$
    $current \leftarrow NULL$
    for $i \leftarrow previous.topLevel - 1$ downto 0 do
        $current \leftarrow previous.next[i]$
        while $current.val < val$ do
            $previous \leftarrow current$
            $current \leftarrow current.next[i]$
        $predecessors[i] \leftarrow previous$
        $successors[i] \leftarrow current$
    $candidate \leftarrow current$
    if $candidate.val == val$ then
        for $i \leftarrow 0$ to $candidate.topLevel - 1$ do
            $predecessors[i].next[i] \leftarrow successors[i].next[i]$
        $FAD(candidate.dataLayer.references)$    ▷ atomically decrement references to data layer node
        return $true$
    return $false$
```

Algorithm 11 Background Process for Index Layer, Consuming Queue and Acting

```
procedure updateNumaZone($DataLayer$)                                              ▷
    $updateQueue \leftarrow DataLayer.updates$
    $sentinel \leftarrow DataLayer.sentinel$
    $runThreadOnNumaZone(DataLayer.numaZone)$          ▷ runs thread on numa zone of the index layer
    while $numask.finished$ is $false$ do
        $operation \leftarrow updateQueue.pop()$
        $executeOperation(sentinel, operation, DataLayer.numaZone)$
```