Latency Hiding in Tree Lookups using Out Of Order Execution

Lukas Karnowski

November 28, 2017
Table of contents

Introduction

Adaptive Radix Tree

Out Of Order Execution

Implementation in the ART

Evaluation

Bibliography / Image Sources
Table of contents

Introduction

Adaptive Radix Tree

Out Of Order Execution

Implementation in the ART

Evaluation

Bibliography / Image Sources
Latency Hiding in Tree Lookups using Out Of Order Execution

What the . . . ?
Latency Hiding in Tree Lookups using Out Of Order Execution

What the . . . ?

Latency Hiding
Latency Hiding in Tree Lookups using Out Of Order Execution

What the . . . ?

Latency Hiding in Tree Lookups
Latency Hiding in Tree Lookups using Out Of Order Execution

What the . . . ?
Table of contents

Introduction

Adaptive Radix Tree

Out Of Order Execution

Implementation in the ART

Evaluation

Bibliography / Image Sources
Adaptive Radix Tree

A\text{daptive} \text{ R} \text{radix} \text{T} \text{ree}
Adaptive Radix Tree

Adaptive Radek Tree

What's so special?
Adaptive Radix Tree

What's so special?

- Improved radix tree (or prefix tree)
Adaptive Radix Tree

Whats so special?
- Improved radix tree (or prefix tree)
- Dynamically adjusts node size
Adaptive Radix Tree

What's so special?

- Improved radix tree (or prefix tree)
- Dynamically adjusts node size
- Can compress paths
Example radix tree

1\textsuperscript{st} Letter

2\textsuperscript{nd} Letter

3\textsuperscript{rd} Letter

Leaf Nodes
Different node types

- Node4
- Node16
- Node48
- Node256
Different node types

- Node4
- Node16
- Node48
- Node256

Example Node4:

<table>
<thead>
<tr>
<th>Keys (1B each)</th>
<th>Pointer (8B each)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 13 42 255</td>
<td>Ptr to 0  Ptr to 13  Ptr to 42  Ptr to 255</td>
</tr>
</tbody>
</table>
Different node types

- Node4
- Node16
- Node48
- Node256

Example Node4:

<table>
<thead>
<tr>
<th>Keys (1B each)</th>
<th>Pointer (8B each)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>13</td>
</tr>
</tbody>
</table>

Lookup using `findChild()`
Lookup algorithm

```c
lookup (node, key, depth) :
  if node == NULL
    return NULL
  if is Leaf (node)
    if leaf Matches (node, key, depth)
      return node
    return NULL
  // ...
  next = find Child (node, key[depth])
  return lookup (next, key, depth + 1)
```
Lookup algorithm

```python
def lookup(node, key, depth):
    if node == NULL
        return NULL
    if isLeaf(node)
        if leafMatches(node, key, depth)
            return node
        return NULL
    // ...
    next = findChild(node, key[depth])
    return lookup(next, key, depth+1)
```
Lookup algorithm

![Diagram of a lookup algorithm with nodes labeled AN and AR, and leaf nodes labeled AND, ANT, ANY, ARE, ART.](image)
What is Out Of Order Execution?

\[(a+b)+(c+d)\]

No dependency between \((a+b)\) and \((c+d)\) → Can be calculated in parallel

Especially helpful for expensive operations, like memory accesses
What is Out Of Order Execution?

(a+b)+(c+d)

No dependency between \((a+b)\) and \((c+d)\)
→ Can be calculated in parallel
What is Out Of Order Execution?

\[(a+b)+(c+d)\]

No dependency between \((a+b)\) and \((c+d)\)
→ Can be calculated in parallel

Especially helpful for expensive operations, like memory accesses
Linked List Experiment

One list

Linked List data type:

```c
struct Node {
    Node *next;
    std::uint8_t data[56];
};
```
Linked List Experiment

One list

Linked List data type:

```c
struct Node {
    Node *next;
    std::uint8_t data[56];
};
```

Iteration:

```c
for (Node *curr = list;
     curr != nullptr;
     curr = curr->next) {
    // Empty body
}
```
Linked List Experiment

One list

Linked List data type:

```c
struct Node {
    Node *next;
    std::uint8_t data[56];
};
```

Iteration:

```c
for (Node *curr = list; curr != nullptr; curr = curr->next) {
    // Empty body
}
```

In Assembler:

```assembly
0x3590: mov (%rax),%rax
0x3593: test %rax,%rax ; depends on the first instr.
0x3596: jne 0x3590
```
Linked List Experiment

Two lists

```c
for (Node *curr1 = list1, *curr2 = list2; curr1 != nullptr && curr2 != nullptr; ) {
    // Empty body
}
```

In Assembler:

```
0x3600 : mov (%rax) ,%rax
0x3603 : mov (%rdx) ,%rdx ; No dependency!
0x3606 : test %rax ,%rax
0x3609 : je 0x3610
0x360b : test %rdx ,%rdx
0x360e : jne 0x3600
0x3610 : . . .
```
Linked List Experiment

Two lists

```c
for (Node *curr1 = list1, *curr2 = list2;
    curr1 != nullptr && curr2 != nullptr;
    curr1 = curr1->next, curr2 = curr2->next) {
    // Empty body
}
```
Linked List Experiment

Two lists

```c
for (Node *curr1 = list1, *curr2 = list2;
     curr1 != nullptr && curr2 != nullptr;
     curr1 = curr1->next, curr2 = curr2->next) {
    // Empty body
}
```

In Assembler:

```assembly
0x3600:  mov  (%rax),%rax
0x3603:  mov  (%rdx),%rdx ; No dependency!
0x3606:  test %rax,%rax
0x3609:  je   0x3610
0x360b:  test %rdx,%rdx
0x360e:  jne  0x3600
0x3610:  ...
```
Linked List Experiment

Results
Linked List Experiment

Results

![Bar chart showing visited list entries per μs for different amounts of parallel lists. The x-axis represents the amount of parallel lists from 1 to 12, and the y-axis shows the visited list entries in units of 10^4. The chart indicates an increase in visited list entries as the number of parallel lists increases.]
Table of contents

Introduction

Adaptive Radix Tree

Out Of Order Execution

Implementation in the ART

Evaluation

Bibliography / Image Sources
Basic idea

Perform multiple lookups at the same time. This technique is called Group Prefetching.
Basic idea

Perform multiple lookups at the same time
Perform multiple lookups at the same time

This technique is called Group Prefetching
Basic idea

Perform multiple lookups at the same time

This technique is called **Group Prefetching**

Keep track of every lookup
Tracking each state

How can we track the state of each lookup?

```cpp
struct GPState {
    std::uint8_t key[8];
    Node* node;
    unsigned depth = 0;
    // ...
    bool finished = false;
};
GPState() : node(nullptr) {}
GPState(Node* node) : node(node) {}
```
Tracking each state

How can we track the state of each lookup?

```cpp
struct GPState {
    std::uint8_t key[8];
    Node *node;

    unsigned depth = 0;
    // ...
    bool finished = false;

    GPState() : node(nullptr) {}
    GPState(Node *node) : node(node) {}
};
```
The actual lookup algorithm
The actual lookup algorithm

```c++
void lookupGP(std::vector<GPState> &states) {
    while (/* not all finished */) {
        // Loop over every state
        for (auto &state : states) {
            if (state.finished)
                continue;

            // Perform the normal lookup algorithm step
            if (state.node == NULL || isLeaf(state.node)) {
                state.finished = true;
                continue;
            }

            state.node = *findChild(state.node, 
                state.key[state.depth]);
            state.depth++;
        }
    }
}
```
Benchmarking

TPC-H benchmark (see e.g. HyperDB Webinterface)

Joining lineitem with orders

lineitem has foreign key to orders

Creating an index on orders

Iterating the tuples in lineitem and performing a lookup in the ART for orders (with multiple keys using GP)

Amount of parallel lookups is called Group Size
TPC-H benchmark (see e.g. HyperDB Webinterface)
Benchmarking

- TPC-H benchmark (see e.g. HyperDB Webinterface)
- Joining lineitem with orders
Benchmarking

- TPC-H benchmark (see e.g. HyperDB Webinterface)
- Joining lineitem with orders
- lineitem has foreign key to orders
- TPC-H benchmark (see e.g. HyperDB Webinterface)
- Joining lineitem with orders
- lineitem has foreign key to orders
- Creating an index on orders
Benchmarking

- TPC-H benchmark (see e.g. HyperDB Webinterface)
- Joining lineitem with orders
- lineitem has foreign key to orders
- Creating an index on orders
- Iterating the tuples in lineitem and performing a lookup in the ART for orders (with multiple keys using GP)
Benchmarking

- TPC-H benchmark (see e.g. HyperDB Webinterface)
- Joining lineitem with orders
- lineitem has foreign key to orders
- Creating an index on orders
- Iterating the tuples in lineitem and performing a lookup in the ART for orders (with multiple keys using GP)
- Amount of parallel lookups is called **Group Size**
Benchmarking Results
Ordered
Benchmarking Results

Ordered

Lookups per \( \mu s \) vs. Group Size

- Regular

\[
\begin{array}{c|c|c|c|c|c|c|c|c|c|c|c}
\text{Group Size} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\
\hline
\text{Lookups per \( \mu s \)} & 2.5 & 3 & 4 & 4.5 & 5 & 5.5 & 6 & 6.5 & 7 & 7.5 & 8 & 8.5 \\
\end{array}
\]
Benchmarking Results

Unordered
Benchmarking Results

Unordered

![Graph showing lookup times for different group sizes.](image)
My reaction
Table of contents

Introduction

Adaptive Radix Tree

Out Of Order Execution

Implementation in the ART

Evaluation

Bibliography / Image Sources
Lessons learned?

Group Prefetching . . .
Lessons learned?

Group Prefetching . . .
- . . . increases Performance, but not as much as seen in the Linked List experiment
Lessons learned?

Group Prefetching . . .

▶ . . . increases Performance, but not as much as seen in the Linked List experiment

▶ . . . gives about 200% speed increase
Lessons learned?

Group Prefetching . . .

- . . . increases Performance, but not as much as seen in the Linked List experiment
- . . . gives about 200% speed increase
- . . . is always useful, when lookup keys are known in advance (e.g. during a Join)
Lessons learned?

Group Prefetching . . .

➤ . . . increases Performance, but not as much as seen in the Linked List experiment

➤ . . . gives about 200% speed increase

➤ . . . is always useful, when lookup keys are known in advance (e.g. during a Join)

➤ . . . can be adjusted using the Group Size variable. Concrete value changes speed increase
→ perfect value depends on use case and hardware
Lessons learned?

Group Prefetching . . .

- . . . increases Performance, but not as much as seen in the Linked List experiment
- . . . gives about 200% speed increase
- . . . is always useful, when lookup keys are known in advance (e.g. during a Join)
- . . . can be adjusted using the Group Size variable. Concrete value changes speed increase
  → perfect value depends on use case and hardware

Out Of Order Execution is quite cool
Latency Hiding in Tree Lookups using Out Of Order Execution
Table of contents

Introduction

Adaptive Radix Tree

Out Of Order Execution

Implementation in the ART

Evaluation

Bibliography / Image Sources


http://i0.kym-cdn.com/entries/icons/mobile/000/001/007/WAT.jpg