CS 310: Memory Hierarchy and B-Trees

Chris Kauffman

Week 14-1
Matrix Sum

Given an M by N matrix \( X \), sum its elements

- M rows, N columns

**Sum R**

```
given X, M, N
sum = 0
for i=0 to M-1{
    for j=0 to N-1 {
        sum += X[i][j]
    }
}
```

**Sum C**

```
given X, M, N
sum = 0
for j=0 to N-1{
    for i=0 to M-1 {
        sum += X[i][j]
    }
}
```

- What’s the difference?
- What’s the complexity of each?
- Should the execution speed be different?
How does a CPU work?

CPU: Sees a load instruction

500:  lw  $t1, $t4
504:  lw  $t2, 4($t4)
508:  add  $t3, $t1, $t2

Load a \textit{word} of memory

- Load value at address in register \texttt{t4} into register \texttt{t1}
- \texttt{ex: t4} contains the memory address 1024, integer 7 is there

Client/Server model

- CPU: requester
- Memory subsystem: provider
- Like you asking for a specific web page
  - Just viewed it a minute ago (fast)
  - GMU web site (medium)
  - Philippines hosted site (slow)
When analyzing code, usually assume *uniform memory access*

- Same time to move any byte/word to a CPU register

Real world: *non-uniform memory access*

- Some memory locations are "farther" away

The memory hierarchy

- Presents a uniform memory access interface
- Tries hard to provide it
- Fails
The Memory Pyramid

Source Article
Edited Excerpt of Jeff Dean’s talk on data centers.

<table>
<thead>
<tr>
<th>Reference</th>
<th>Time</th>
<th>Analogy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>-</td>
<td>Your brain</td>
</tr>
<tr>
<td>L1 cache reference</td>
<td>0.5 ns</td>
<td>Your desk</td>
</tr>
<tr>
<td>L2 cache reference</td>
<td>7 ns</td>
<td>Neighbor’s Desk</td>
</tr>
<tr>
<td>Main memory reference</td>
<td>100 ns</td>
<td>This Room</td>
</tr>
<tr>
<td>Disk seek</td>
<td>10,000,000 ns</td>
<td>Salt Lake City</td>
</tr>
</tbody>
</table>

Does Big-O analysis capture these effects?
What’s a Cache

500: lw $t1, $t4

t4 contains address 1024, lw moves word at 1024 into register t1

Side-effect

▶ Memory addresses "around" 1024 are loaded into cache
▶ Probably something like addresses 1024 to 2047 (1K) end up in L1 cache
▶ Referred to as a cache line
▶ Subsequent accesses to 1028, 1032, ... 2044 will happen fast

Cache is a limited resource

▶ Putting one line in cache overwrites another line
▶ Later load address 5120, 1024-2047 evicted from cache
Cache Affects Performance

As measured by hardware counters using Linux’s `perf` on:

- Model name: Intel(R) Core(TM) i5-2400 CPU @ 3.10GHz
- Cache size: 6144 KB

With

```
> perf stat $opts java MatrixSums 8000 4000 row
> perf stat $opts java MatrixSums 8000 4000 col
```

<table>
<thead>
<tr>
<th>Measurement</th>
<th>row</th>
<th>col</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cycles</td>
<td>3,507,364,715</td>
<td>5,605,621,966</td>
</tr>
<tr>
<td>Instructions</td>
<td>2,353,887,029</td>
<td>2,543,165,478</td>
</tr>
<tr>
<td>L1-dcache-loads</td>
<td>527,694,054</td>
<td>561,540,169</td>
</tr>
<tr>
<td>L1-dcache-load-misses</td>
<td>25,638,014</td>
<td>122,663,199</td>
</tr>
<tr>
<td>Runtime (seconds)</td>
<td>1.001</td>
<td>1.620</td>
</tr>
</tbody>
</table>

L1 data cache load misses

- Row: 25K/548K = 4% main memory access
- Col: 122/585K = 20% main memory access
Cache and Main Memory

**Concern**
Binary search trees don’t focus on exploiting cache very much
- Left/Right in cache 1-7ns access time
- Left/Right not in cache, 100ns trip to main memory
- Could do 100 operations during that trip (!)

**Problem**
- Left/Right not in main memory, 10,000,000 ns trip to disk
  - CPU gets a siesta, user gets irate
- When would this happen?
Big Data

- Machine named HAL has
  - 1mb cache
  - 8gb memory
- Database DB has
  - record size $2^{11}$ b (2048 bytes, 2kb)
  - $2^{24}$ records (16 mb)
  - Size: $2^{24} \times 2^{11}$ b = $2^{35}$ b (32gb)
- Find Record R in DB stored on HAL

**Bad implementation**
Store DB randomly, search sequentially for R
- Is the DB any bigger this way?

**A bit better**
Store DB as a balanced BST, binary searches for R
- How big is the new DB with left/right pointers?
- How deep is the tree?
- How many disk accesses may be needed?
Deep Trees

- Database DB has
  - record size $2^{11}$ b (2048 bytes, 2kb)
  - $2^{24}$ records (16 mb)
  - Size: $2^{24} \times 2^{11}$ b = $2^{35}$ b (32gb)
- Find Record Z in Y stored on X
- Store DB in single BST, use binary search for R

Answers

- How big is the new DB with left/right pointers?
  - $2^{24}$ records
  - $2 \times 8$ b pointers per record for left/right = 16b per record = $2^4$ b
  - $2^{24} \times 2^4 = 2^{28}$ b = 256mb
  - Small compared to 32gb (0.7%)
- How deep is the tree?
  - $2^{24}$ records, log$_2$, expect 24 deep
- How many disk accesses may be needed?
  - Very unlucky - 24 accesses
  - Each costs 10,000,000 ns
  - Could have done 240,000,000 instructions
Tree + Array = B-Tree

Large DB’s use sequential ordering with gaps, tree index
- Sequential chunks allow array-searching in cache
- Whole index doesn’t fit in fast memory, but chunks do
- Do as much work as possible in fast memory to avoid slow disk access

B-trees exploit this to reduce tree depth / disk accesses

Internal Nodes
- Branch more than 2 ways
- Store multiple keys
- Keys in a sorted array
- Make sure they fit in cache
- Use a sequential search to find branch
- Always half full to full
  - root exception

Leaves
- Data is only at the leaves
- Hold multiple sorted data
- Have maximum data capacity
- Optimized to disk block size
- Always half full to full
The origin of "B-tree" has never been explained by the authors. As we shall see, "balanced," "broad," or "bushy" might apply. Others suggest that the "B" stands for Boeing. Because of his contributions, however, it seems appropriate to think of B-trees as "Bayer"-trees.
– Wikipedia: B-tree
B-Trees Ops

Original

Insert 57
B-Trees Ops

Inserted 57

Insert 55
B-Trees Ops

Inserted 55

Insert 40
B-Trees Ops

Inserted 40

Delete 99
General Strategies

**ADD() quasi-code**

ADD(x, bt)
    find right leaf in bt
    if space in leaf
        add x to leaf
    else
        if parent has room
            new leaf
            split data
            add x to leaf
        else
            recurse up
            split internal
            new leaves
            split data
            back down to add x

**REMOVE() quasi-code**

REMOVE(x, bt)
    find leaf with x
    remove x
    if leaf < 1/2 full
        merge with neighbor leaf
        steal leaves if needed
        recurse up to adjust
B-tree Take-home

- Multi-way trees
- If order-\(k\) nodes are all \(1/2\) full \(\rightarrow O(\log_k N)\) height
- Hybrid of array/tree
- Good for data that doesn’t fit in memory
  - Large Databases
  - Filesystems
  - Sensitive to memory hierarchy
- Simple idea, complex implementation
- Many variations on the idea
- No Weiss B-trees: too complex