Memory Optimization

Some slides from Christer Ericson
Sony Computer Entertainment, Santa Monica
Overview

► We have seen how to reorganize matrix computations to improve temporal and spatial locality
  ▪ Improving spatial locality required knowing the layout of the matrix in memory
► Orthogonal approach
  ▪ Change the representation of the data structure in memory to improve locality for a given pattern of data accesses from the computation
  ▪ Less theory exists for this but some nice results are available for trees: van Emde Boas tree layout
► Similar ideas can be used for graph algorithms as well
  ▪ However there is usually not as much locality in graph algorithms
Data cache optimization

► Compressing data
► Prefetching data into cache
► Cache-conscious data structure layout
  ▪ Tree data structures
► Linearization caching
Prefetching

Software prefetching
- Not too early – data may be evicted before use
- Not too late – data not fetched in time for use
- Greedy

Instructions
- iA-64: lfetch (line prefetch)
  - Options:
    - Intend to write: begins invalidations in other caches
    - Which level of cache to prefetch into
- Compilers and programmers can access through intrinsics
const int kLookAhead = 4; // Some elements ahead
for (int i = 0; i < 4 * n; i += 4) {
    Prefetch(elem[i + kLookAhead]);
    Process(elem[i + 0]);
    Process(elem[i + 1]);
    Process(elem[i + 2]);
    Process(elem[i + 3]);
}

// Loop through and process all 4n elements
for (int i = 0; i < 4 * n; i++)
    Process(elem[i]);
void PreorderTraversal(Node *pNode) {
    // Greedily prefetch left traversal path
    Prefetch(pNode->left);
    // Process the current node
    Process(pNode);
    // Greedily prefetch right traversal path
    Prefetch(pNode->right);
    // Recursively visit left then right subtree
    PreorderTraversal(pNode->left);
    PreorderTraversal(pNode->right);
}
Data structure representation

► Cache-conscious layout
  ▪ Node layout
    ► Field reordering (usually grouped conceptually)
    ► Hot/cold splitting
  ▪ Overall data structure layout

► Little compiler support
  ▪ Easier for non-pointer languages (Java)
  ▪ C/C++: do it yourself
Field reordering

```c
struct S {
    void *key;
    int count[20];
    S *pNext;
};

void Foo(S *p, void *key, int k) {
    while (p) {
        if (p->key == key) {
            p->count[k]++;
            break;
        }
        p = p->pNext;
    }
}
```

► Likely accessed together so store them together!
Hot/cold splitting

Hot fields:

```c
struct S {
    void *key;
    S *pNext;
    S2 *pCold;
};
```

Cold fields:

```c
struct S2 {
    int count[10];
};
```

- Split cold fields into a separate structure
- Allocate all `struct S` from a memory pool
  - Increases coherence
Tree data structures

- **Rearrange nodes**
  - Increase spatial locality
  - Cache-aware vs. cache-oblivious layouts

- **Reduce size**
  - Pointer elimination (using implicit pointers)
  - “Compression”
    - Quantize values
    - Store data relative to parent node
Definitions:

A tree $T_1$ can be **embedded** in another tree $T_2$, if $T_1$ can be obtained from $T_2$ by pruning subtrees.

**Implicit layout** - the navigation between a node and its children is done based on address arithmetic, and not on pointers.
Breadth-first order

- Pointer-less: $\text{Left}(n)=2n$, $\text{Right}(n)=2n+1$
- Requires storage for complete tree of height $H$
Depth-first order

- Left(n) = n + 1, Right(n) = stored index
- Only stores existing nodes
Cache Oblivious Binary Search Trees

Gerth Stolting Brodal
Rolf Fagerberg
Riko Jacob
Motivation

➢ Our goal:
  • To find an implementation for binary search tree that tries to minimize cache misses.
  • That algorithm will be cache oblivious.

➢ By optimizing an algorithm to one unknown memory level, it is optimized to each memory level automatically!
Assume we have a binary search tree.
Embed this tree in a static complete tree.
Save this (complete) tree in the memory in a cache oblivious fashion
- Complete tree permits storing the tree without child pointers
  - However there may be some empty subtrees
On insertion, create a new static tree of double the size if needed.
General idea & Working methods

Advantages:

- Minimizing memory transfers.
- Cache obliviousness
- No pointers – better space utilization:
  - A larger fraction of the structure can reside in lower levels of the memory.
  - More elements can fit in a cache line.

Disadvantages:

- Implicit layout: higher instruction count per navigation – slower.
van Emde Boas memory layout

- Recursive definition:
- A tree with only one node is a single node record.
- If a tree $T$ has two or more nodes:
  - Divide $T$ to a top tree $T_0$ with height $[h(T)/2]$ and a collection of bottom trees $T_1, ..., T_k$ with height $[h(T)/2]$ , numbered from left to right.
  - The van Emde Boas layout of $T$ consist of the v.E.B. layout of $T_0$ followed by the v.E.B. layout of $T_1, ..., T_k$
van Emde Boas memory layout

Example:
van Emde Boas memory layout

Example:
van Emde Boas memory layout

Example:
van Emde Boas memory layout

Example:
van Emde Boas memory layout

Example:

```
    1
   / \
  2   3
```

---

23
van Emde Boas memory layout

Example:
van Emde Boas memory layout

Example:
van Emde Boas memory layout

Example:
van Emde Boas memory layout

Example:

```
1 2 3
5 6 7 8 9
```
van Emde Boas memory layout

Example:

```
  1
 / \
2   3
 / \
4   5
 / \
6   7
 / \
8   9
 / \
10 11
 / \
12
```
van Emde Boas memory layout

Example:

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
```
The algorithm

- **Search:**
  - Standard search in a binary tree.
  - Memory transfers: $O(\log_B n)$ worst case

- **Range query:**
  - Standard range query in a binary tree:
    - Search the smallest element in the range
    - Make an inorder traversals till you reach an element greater then or equals to the greatest element in the range.
  - Memory transfers: $O(\log_B n + k/B)$ worst case
Insertions

Intuitive idea:

- Locate the position in T of the new node (regular search)
- If there is an empty slot there, just insert the new value there
- If tree has some empty slots, rebalance T and then insert the new value
- Otherwise, use recursive doubling
  - Allocate a new tree for double the depth of the current tree
  - Copy over values from new tree to old tree
Example: insert 7

\[ \rho = 0.71 \]

\[ \rho = 0.33 \]
Example: insert 7

\[ \rho = 1 \]

\[ \rho = 0.71 \]

\[ \rho = 0.33 \]
Rebalancing

Example: insert 7

\[
\begin{align*}
\rho &= 0.71 \\
\rho &= 0.33 \\
\rho &= 1
\end{align*}
\]
Rebalancing

Example: insert 7

\[ \rho = 1 \]

\[ \rho = 0.71 \]

\[ \rho = 0.33 \]
Rebalancing

Example:

Insert 7

$\rho = 0.71$

$\rho = 0.33$
Rebalancing

Example: insert 7

\[ \rho = 0.71 \]

\[ \rho = 0.33 \]
Example: insert 7

The next insertion will cause a rebuilding
Linearization caching

- **Nothing better than linear data**
  - Best possible spatial locality
  - Easily prefetchable

- **So linearize data at runtime!**
  - Fetch data, store linearized in a custom cache
  - Use it to linearize...
    - hierarchy traversals
    - indexed data
    - other random-access stuff
Leaves: $f_0, f_1, f_2, f_0, f_1, f_7, f_8, f_1, f_4, f_5, f_7, \ldots$

Faces: $v_0, v_2, v_3, v_0, v_4, v_5, v_1, v_2, v_4, \ldots$

Vertices: $(x_0, y_0, z_0), (x_1, y_1, z_1), (x_2, y_2, z_2), (x_3, y_3, z_3), (x_4, y_4, z_4), \ldots$

Cached linearized leaf: $(x_0, y_0, z_0), (x_2, y_2, z_2), (x_3, y_3, z_3), (x_0, y_0, z_0), (x_4, y_4, z_4), \ldots$
Relating graphs and matrices

• Graphs can be viewed as matrices and vice versa
• Order of edge visits in algorithm = order of matrix entry visits
  – Row-wise traversal of matrix = visit each node of graph and walk over its outgoing edges
  – Column-wise traversal of matrix = visit each node of graph and walk over its incoming edges
  – Block traversal of matrix = ?
Locality in ADP model

- **Temporal locality:**
  - Activities with overlapping neighborhoods should be scheduled close together in time on same core
  - Example: activities i₁ and i₂

- **Spatial locality:**
  - Abstract view of graph can be misleading
  - Depends on the concrete representation of the data structure

- **Inter-package locality:**
  - Partition graph between packages and partition concrete data structure correspondingly (see next time)
  - Active node is processed by package that owns that node

Abstract data structure

<table>
<thead>
<tr>
<th>src</th>
<th>1</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>dst</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>val</td>
<td>3.4</td>
<td>3.6</td>
<td>0.9</td>
<td>2.1</td>
</tr>
</tbody>
</table>
Galois Graph

- Local computation graph:
  - Compressed sparse row (CSR) storage permits exploitation of temporal and spatial locality for algorithms that iterate over edges of a given node
  - More compact versions that inline some of the arrays in CSR format are also available

Compressed sparse row (CSR)  More compact representations
Summary
Friends: The 3 R’s

► **Rearrange (code, data)**
  - Change layout to increase spatial locality

► **Reduce (size, # cache lines read)**
  - Smaller/smarter formats, compression

► **Reuse (cache lines)**
  - Increase temporal (and spatial) locality

<table>
<thead>
<tr>
<th></th>
<th>Compulsory</th>
<th>Capacity</th>
<th>Conflict</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rearrange</td>
<td>X</td>
<td>(x)</td>
<td>X</td>
</tr>
<tr>
<td>Reduce</td>
<td>X</td>
<td>X</td>
<td>(x)</td>
</tr>
<tr>
<td>Reuse</td>
<td>(x)</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>