Topics

- Field reordering
- Structure splitting
- Structure pealing
  - SoA v.s. AoS
- Base+offset addressing
- Pointer Compression
- Global Variable Ordering
- Memory Hierarchy Aware Recursive layout
- Allocation
  - Coallocation
  - Granularity

Field Reordering

- Change the order of fields in a structure to improve cache behavior
- How to pick fields to reorder?

```
struct a {
  char c;
  int i;
  double d;
  float f;
  *ptr;
}
```

```
struct a {
  double d;
  float f;
  char c;
  *ptr;
  int i;
}
```

Hot/Cold

- Hot code: loops/region of code where the most time is being spent
- Hot fields: fields used in hot code
- Reorder to cluster hot fields together at beginning of structure

```
struct a {
  char c;
  int i; //HOT
  double d;
  float f;
  *ptr; //HOT
}
```

```
struct a {
  int i;
  char c;
  *ptr;
  double d;
  float f;
}
```

Hot/Cold

- Increase likelihood of sharing cacheline
- Can repeat process for different hot regions

```
struct a {
  char c;
  short s;
  int i;
  float f;
  double d;
  *ptr; //HOT A,B
}
```

```
struct a {
  char c;
  int i;
  short s;
  double d;
  float f;
  *ptr; //HOT A,B
}
```

Padding Elimination

- Reorder struct to eliminate padding
- Small size, denser packing, more items fit in cache

```
struct a {
  char c;
  short s;
  int i;
  float f;
  double d;
  *ptr;
}
```

```
struct a {
  char c;
  short s;
  int i;
  float f;
  double d;
  *ptr;
}
```

```
struct a {
  char c;
  short s;
  int i;
  float f;
  double d;
  *ptr;
}
```

```
struct a {
  char c;
  short s;
  int i;
  float f;
  double d;
  *ptr;
}
```

```
struct a {
  char c;
  short s;
  int i;
  float f;
  double d;
  *ptr;
}
```

```
struct a {
  char c;
  short s;
  int i;
  float f;
  double d;
  *ptr;
}
```

```
struct a {
  char c;
  short s;
  int i;
  float f;
  double d;
  *ptr;
}
```

```
struct a {
  char c;
  short s;
  int i;
  float f;
  double d;
  *ptr;
}
```

```
struct a {
  char c;
  short s;
  int i;
  float f;
  double d;
  *ptr;
}
```

```
struct a {
  char c;
  short s;
  int i;
  float f;
  double d;
  *ptr;
}
```

```
struct a {
  char c;
  short s;
  int i;
  float f;
  double d;
  *ptr;
}
```

```
struct a {
  char c;
  short s;
  int i;
  float f;
  double d;
  *ptr;
}
```
Read/Write

- Group Written fields on a single cache line
- Writes go to memory by unit of cache line

```
struct s {
  int i;       //W
  double d;    //W
  float f;
  char c[64];
  s* ptr;
};
```

Structure Splitting

- Take one structure and make it two:
  - Hot structure + pointer
  - Cold structure
- Operations on hot structure are faster, operations on cold structure are slower

```
struct s {
  int i;
  double d;
  char c[64];
  float f;
  s* ptr;
};    //96 bytes
```
```
struct s {
  double d;
  s* ptr;
  sc* rest;
};    //24 bytes      //72 bytes
```
```
struct sc {
  int i;
  char c[64];
  float f;
};    //72 bytes
```

- Usually this takes more space

Structure pealing

- Special case of splitting
- When we can find the cold portion without a pointer, we can omit the pointer

```
struct s {
  int i;
  char c[64];
  double d;
  float f;
  s* ptr;
};    //96 bytes
```
```
struct s {
  double d;
  s* ptr;
};    //16 bytes
```
```
struct sc {
  int i;
  char c[64];
  float f;
};    //72 bytes
```

- Usually this takes the same space

AoS v.s. SoA

- Special case of pealing

```
struct s {
  int i;
  char c[64];
  double d;
  float f;
  s* ptr;
};
```
```
struct s a[128];
```
```
struct a {
  int i[128];
  char c[128][64];
  double d[128];
  s* ptr[128];
};
```
```
struct s a;
```
```
struct sc ac;
```
```
struct s ac[16];
```

Choosing the Transformation

- Sizeof(struct) << cache block size
  - No action
- Sizeof(struct) ~= cache block size
  - Split structure
- Sizeof(struct) >> cache block size
  - Reorder fields
- Parallel
  - Above + Another lecture
- Vectorizing
  - SoA or field reordering
Base + offset Addressing

- Replace pointers with smaller data types
  - Rather than pointer to object, store offset from an otherwise known base pointer
- Limits data structure size

Base + Offset example

- Consider a set implemented as a red black tree
  - Each node has an element and two pointers
  - Nodes are allocated via new/malloc
- Properties
  - Nodes and allocations invisible to the programmer
  - Nodes never escape
- Replace heap allocation with allocation in a vector or deque and replace pointers with uint16 offsets
  - Overhead 16B -> 4B
  - Max $2^{16}$ entries in set

Structural Addressing

- Make pointers implicit in recursive data structures
- Pointers can be computed based only on the address of the current node

Structural Addressing – Binary Tree

```c
struct node {
    T data;
    node* c[2];
};

node* find (node* tree, T data) {
    if (tree->data == data)
        return tree;
    return find(tree->c[tree->data > data]);
}

T* find (std::vector<node> tree, T data, int offset = 0) {
    if (tree[offset] == data)
        return &tree[offset];
    return find(tree, data, offset * 2 + 1 + (tree[offset] > data));
}
```

Node I has children at $I\times2 + 1$ and $I\times2 + 2$

Pointer Compression

- Newer instruction sets introduced useful performance features when transitioning from 32->64
  - amd64 has more registers than x86
    - But each instruction takes more space
    - But average program length is less
    - But average program is faster
  - SparcV9: cmov, more registers, better branches
Pointer Compression

- Increase in address space and pointer size makes programs slower
  - A binary tree may take almost 2x memory

*Use smaller pointers*
(still run in new, faster mode)

**Pointer Compression**

- ABI based
  - X32 abi (Linux x86, similar thing on sparc)
- Compiler Based
  - Transparent Pointer Compression
  - Base + offset addressing by type
- Exotic hardware
  - Architecture students need PhDs too

**Lattner, MSP05**

- [Graph showing data]

**Global Variable Ordering**

- Globals are layed out in memory by the linker
- You can think of globals as fields in one global structure
- Thus we expect reordering globals to be advantageous

**Global Variable Ordering**

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Global Variable Ordering</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>% improvement for 800</td>
</tr>
<tr>
<td>MSP05</td>
<td>0.52</td>
</tr>
<tr>
<td></td>
<td>0.52</td>
</tr>
<tr>
<td></td>
<td>0.52</td>
</tr>
<tr>
<td></td>
<td>0.52</td>
</tr>
</tbody>
</table>

**Allocation**

- Memory aware recursive layout
- Coallocation
- Granularity
Memory Aware Recursive Layout

- Read “You’re Doing it Wrong”, Poul-Henning Kamp, Communications of the ACM, July 2010
- You want to block your recursive data structures for memory
- Ideally the next visited node is in the same cacheline or page as the current node

Consider the Binary Tree

Consider the Binary Tree
Implemented Using a Heap

Memory Page Layout of Heap

Layout of B-Heap, blocked for pages

Doesn’t fit in memory, fast disk

More complex indexing function
Coallocation

- Dynamic allocations look a lot like an array of variable sized objects
  - Especially in gc languages
- If we know which objects are accessed near each other, we can allocate them near each other
- Like structure pealing

Coallocation ala MSR

Granularity

- Allocating small objects wastes space
  - Overhead per object in malloc (4-8 bytes)
  - Padding for alignment
- Allocating an array of objects likely uses less memory than allocating each individual object

Sub allocators

- Constrained behavior allows optimization
- Set of objects dead after known point?
  - Region allocator
- Objects freed in reverse order to allocation?
  - Obstack allocator
- All allocations the same size?
  - Fixed sized block allocator

Automatic Pool Allocation

- Compiler builds allocator for specific instances of data structures
- Optimized allocator and allocations based on usage

Lattner PLDI05
Continued, Optimized Allocators

Pool optimizations help some progs that pool allocation itself doesn’t

Continued, access patterns

Continued, access patterns

Continued, access patterns