Automatic Pool Allocation: Improving Performance by Controlling Data Structure Layout in the Heap

Chris Lattner  
lattner@cs.uiuc.edu

Vikram Adve  
vadve@cs.uiuc.edu

June 13, 2005
PLDI 2005

http://llvm.cs.uiuc.edu/
What is the problem?

- List 1 Nodes
- List 2 Nodes
- Tree Nodes

What the compiler sees:

What we want the program to create and the compiler to see:
Our Approach: Segregate the Heap

- **Step #1: Memory Usage Analysis**
  - Build context-sensitive points-to graphs for program
  - We use a fast unification-based algorithm

- **Step #2: Automatic Pool Allocation**
  - Segregate memory based on points-to graph nodes
  - Find lifetime bounds for memory with escape analysis
  - Preserve points-to graph-to-pool mapping

- **Step #3: Follow-on pool-specific optimizations**
  - Use segregation and points-to graph for later optzns
Why Segregate Data Structures?

**Primary Goal: Better compiler information & control**
- Compiler knows where each data structure lives in memory
- Compiler knows order of data in memory (in some cases)
- Compiler knows type info for heap objects (from points-to info)
- Compiler knows which pools point to which other pools

**Second Goal: Better performance**
- Smaller working sets
- Improved spatial locality
- Sometimes convert irregular strides to regular strides
Contributions

1. First “region inference” technique for C/C++:
   - Previous work *required* type-safe programs: ML, Java
   - Previous work focused on memory management

2. Region inference driven by pointer analysis:
   - Enables handling non-type-safe programs
   - Simplifies handling imperative programs
   - Simplifies further pool+ptr transformations

3. New pool-based optimizations:
   - Exploit per-pool and pool-specific properties

4. Evaluation of impact on memory hierarchy:
   - We show that pool allocation reduces working sets
Talk Outline

- Introduction & Motivation
- **Automatic Pool Allocation Transformation**
- Pool Allocation-Based Optimizations
- Pool Allocation & Optzn Performance Impact
- Conclusion
Automatic Pool Allocation Overview

- Segregate memory according to points-to graph
- Use context-sensitive analysis to distinguish between RDS instances passed to common routines

Points-to graph (two disjoint linked lists)

Pool 1

Pool 2

Pool 3

Pool 4
**Points-to Graph Assumptions**

- **Specific assumptions:**
  - Build a points-to graph for each function
  - Context sensitive
  - Unification-based graph
  - Can be used to compute escape info

- **Use any points-to that satisfies the above**

- **Our implementation uses DSA [Lattner:PhD]**
  - Infers C type info for many objects
  - Field-sensitive analysis
  - Results show that it is very fast
Pool Allocation: Example

```c
list *makeList(int Num, Pool* P) {
    list *New = poolalloc(P);
    New->Next = Num ? makeList(Num - 1, P) : 0;
    New->Data = Num; return New;
}

int twoLists(Pool* P2) {
    Pool P1; poolinit(&P1);
    list *X = makeList(10, &P1);
    list *Y = makeList(100, P2);
    GL = Y;
    processList(X);
    processList(Y);
    freeList(X, &P1);
    freeList(Y, P2);
    pooldestroy(&P1);
}
```

Change calls to free into calls to poolfree \(\rightarrow\) retain explicit deallocation

P1

P2

int: GMRC Global

list: HMRC

list*

int

list:

list*

int

list:

list*
Pool Allocation Algorithm Details

- **Indirect Function Call Handling:**
  - Partition functions into equivalence classes:
    - If F1, F2 have *common call-site* ⇒ same class
  - Merge points-to graphs for each equivalence class
  - *Apply previous transformation unchanged*

- **Global variables pointing to memory nodes**
  - See paper for details

- **poolcreate/pooldestroy placement**
  - See paper for details
Talk Outline

- Introduction & Motivation
- Automatic Pool Allocation Transformation
- Pool Allocation-Based Optimizations
- Pool Allocation & Optzn Performance Impact
- Conclusion
Pool Specific Optimizations

**Different Data Structures Have Different Properties**

- **Pool allocation segregates heap:**
  - Roughly into logical data structures
  - Optimize using pool-specific properties

**Examples of properties we look for:**
- Pool is type-homogenous
- Pool contains data that only requires 4-byte alignment
- Opportunities to reduce allocation overhead
Looking closely: Anatomy of a heap

- **Fully general malloc-compatible allocator:**
  - Supports malloc/free/realloc/memalign etc.
  - Standard malloc overheads: object header, alignment
  - Allocates slabs of memory with exponential growth
  - By default, all returned pointers are 8-byte aligned

- **In memory, things look like (16 byte allocs):**

  ![Diagram of memory allocation]

  - One 32-byte Cache Line
PAOpts (1/4) and (2/4)

- Selective Pool Allocation
  - Don’t pool allocate when not profitable

- PoolFree Elimination
  - Remove explicit de-allocations that are not needed

See the paper for details!
PAOpt (3/4): Bump Pointer Optzn

- If a pool has no poolfree’s:
  - Eliminate per-object header
  - Eliminate freelist overhead (faster object allocation)
- Eliminates 4 bytes of inter-object padding
  - Pack objects more densely in the cache
- Interacts with poolfree elimination (PAOpt 2/4)!
  - If poolfree elim deletes all frees, BumpPtr can apply

<table>
<thead>
<tr>
<th>16-byte user data</th>
<th>16-byte user data</th>
<th>16-byte user data</th>
<th>16-byte user data</th>
</tr>
</thead>
</table>

One 32-byte Cache Line

Chris Lattner
PAOpts (4/4): Alignment Analysis

- **Malloc must return 8-byte aligned memory:**
  - It has no idea what types will be used in the memory
  - Some machines bus error, others suffer performance problems for unaligned memory

- **Type-safe pools infer a type for the pool:**
  - Use 4-byte alignment for pools we know don’t need it
  - Reduces inter-object padding

---

One 32-byte Cache Line

<table>
<thead>
<tr>
<th>16-byte user data</th>
<th>16-byte user data</th>
<th>16-byte user data</th>
</tr>
</thead>
</table>

4-byte object header
Talk Outline

- Introduction & Motivation
- Automatic Pool Allocation Transformation
- Pool Allocation-Based Optimizations
- Pool Allocation & Optzn Performance Impact
- Conclusion
### Simple Pool Allocation Statistics

DSA is able to infer that most static pools are type-homogenous suites, plus unbundled programs.

<table>
<thead>
<tr>
<th>Program</th>
<th>LOC</th>
<th>Stat Pools</th>
<th>Num TH</th>
<th>TH%</th>
<th>Dyn Pools</th>
</tr>
</thead>
<tbody>
<tr>
<td>164.gzip</td>
<td>8616</td>
<td>4</td>
<td>4</td>
<td>100%</td>
<td>44</td>
</tr>
<tr>
<td>175.vpr</td>
<td>17728</td>
<td>107</td>
<td>91</td>
<td>85%</td>
<td>44</td>
</tr>
<tr>
<td>197.parser-b</td>
<td>11204</td>
<td>49</td>
<td>48</td>
<td>98%</td>
<td>6674</td>
</tr>
<tr>
<td>252.eon</td>
<td>35819</td>
<td>124</td>
<td>123</td>
<td>99%</td>
<td>66</td>
</tr>
<tr>
<td>300.twolf</td>
<td>20461</td>
<td>94</td>
<td>88</td>
<td>94%</td>
<td>227</td>
</tr>
<tr>
<td>anagram</td>
<td>650</td>
<td>4</td>
<td>3</td>
<td>75%</td>
<td>4</td>
</tr>
<tr>
<td>bc</td>
<td>7297</td>
<td>24</td>
<td>22</td>
<td>91%</td>
<td>19</td>
</tr>
<tr>
<td>ft</td>
<td>1803</td>
<td>3</td>
<td>3</td>
<td>100%</td>
<td>4</td>
</tr>
<tr>
<td>ks</td>
<td>782</td>
<td>3</td>
<td>3</td>
<td>100%</td>
<td>3</td>
</tr>
<tr>
<td>yacr2</td>
<td>3982</td>
<td>20</td>
<td>20</td>
<td>100%</td>
<td>83</td>
</tr>
<tr>
<td>analyzer</td>
<td>923</td>
<td>5</td>
<td>5</td>
<td>100%</td>
<td>8</td>
</tr>
<tr>
<td>neural</td>
<td>785</td>
<td>5</td>
<td>5</td>
<td>100%</td>
<td>93</td>
</tr>
<tr>
<td>pcompress2</td>
<td>903</td>
<td>5</td>
<td>5</td>
<td>100%</td>
<td>8</td>
</tr>
<tr>
<td>llu-bench</td>
<td>191</td>
<td>1</td>
<td>1</td>
<td>100%</td>
<td>2</td>
</tr>
<tr>
<td>chomp</td>
<td>424</td>
<td>4</td>
<td>4</td>
<td>100%</td>
<td>7</td>
</tr>
<tr>
<td>fpgrowth</td>
<td>634</td>
<td>6</td>
<td>6</td>
<td>100%</td>
<td>3.4M</td>
</tr>
<tr>
<td>espresso</td>
<td>14959</td>
<td>160</td>
<td>160</td>
<td>100%</td>
<td>100K</td>
</tr>
<tr>
<td>povray31</td>
<td>108273</td>
<td>46</td>
<td>5</td>
<td>11%</td>
<td>14</td>
</tr>
</tbody>
</table>

DSA + Pool allocation compile time is small: less than 3% of GCC compile time for all tested programs. See paper for details.
Several programs unaffected by pool allocation (see paper)
Sizable speedup across many pointer intensive programs
Some programs (ft, chomp) order of magnitude faster

See paper for control experiments (showing impact of pool runtime library, overhead induced by pool allocation args, etc)
Pool optimizations help some progs that pool allocation itself doesn’t.

- Optimizations help all of these programs:
  - Despite being very simple, they make a big impact.
**Cache/TLB miss reduction**

Miss rate measured with perfctr on AMD Athlon 2100+

Sources:
- Defragmented heap
- Reduced inter-object padding
- Segregating the heap!
Chomp Access Pattern with Malloc

Allocates three object types (red, green, blue)

Spends most time traversing green/red nodes

Each traversal sweeps through all of memory

Blue nodes are interspersed with green/red nodes
Chomp Access Pattern with PoolAlloc
Heap segregation has a similar effect on FT:

- See my Ph.D. thesis for details
Related Work

- **Heuristic-based collocation & layout**
  - Requires programmer annotations or GC
  - Does not segregate based on data structures
  - Not rigorous enough for follow-on compiler transforms

- **Region-based mem management for Java/ML**
  - Focused on replacing GC, not on performance
  - Does not handle weakly-typed languages like C/C++
  - Focus on careful placement of region create/destroy

- **Complementary techniques:**
  - Escape analysis-based stack allocation
  - Intra-node structure field reordering, etc

Chris Lattner
Pool Allocation Conclusion

**Goal of this paper: Memory Hierarchy Performance**

**Two key ideas:**

1. **Segregate heap based on points-to graph**
   - Give compiler some control over layout
   - Give compiler information about locality
   - Context-sensitive $\Rightarrow$ segregate rds instances

2. **Optimize pools based on per-pool properties**
   - Very simple (but useful) optimizations proposed here
   - Optimizations could be applied to other systems

http://llvm.cs.uiuc.edu/
How can you use Pool Allocation?

- **We have also used it for:**
  1. Node collocation & several refinements (this paper)
  2. Memory safety via homogeneous pools [TECS 2005]
  3. 64-bit to 32-bit Pointer compression [MSP 2005]

- **Segregating data structures could help in:**
  - Checkpointing
  - Memory compression
  - Region-based garbage collection
  - Debugging & Visualization
  - More novel optimizations

http://llvm.cs.uiuc.edu/