Transparent Pointer Compression

for Linked Data Structures

June 12, 2005

Vikram Adve
vadve@cs.uiuc.edu

Chris Lattner
lattner@cs.uiuc.edu

June 12, 2005
MSP 2005

http://llvm.cs.uiuc.edu/
Growth of 64-bit computing

- 64-bit architectures are increasingly common:
  - New architectures and chips (G5, IA64, X86-64, …)
  - High-end systems have existed for many years now

- 64-bit address space used for many purposes:
  - Address space randomization (security)
  - Memory mapping large files (databases, etc)
  - Single address space OS’s
  - Many 64-bit systems have < 4GB of phys memory
    - 64-bits is still useful for its virtual address space
Cost of a 64-bit virtual address space

Pointers must be 64 bits (8 bytes) instead of 32 bits:
- Significant impact for pointer-intensive programs!

Pointer intensive programs suffer from:
- Reduced effective L1/L2/TLB cache sizes
- Reduced effective memory bandwidth
- Increased alignment requirements, etc

Pointer intensive programs are increasingly common:
- Recursive data structures (our focus)
- Object oriented programs
Previously Published Approaches

- Simplest approaches: Use 32-bit addressing
  - Compile for 32-bit pointer size “-m32”
  - Force program image into 32-bits [Adl-Tabatabai’04]
  - Loses advantage of 64-bit address spaces!

- Other approaches: Exotic hardware support
  - Compress pairs of values, speculating that pointer offset is small [Zhang’02]
  - Compress arrays of related pointers [Takagi’03]
  - Requires significant changes to cache hierarchy

No previous fully-automatic compiler technique to shrink pointers in RDS’s
1. Use Automatic Pool Allocation [PLDI'05] to partition heap into memory pools:
   - Infers and captures pool homogeneity information
Our Approach (2/2)

2. Replace pointers with 64-bit integer offsets from the start of the pool
   - Change *Ptr into *(PoolBase+Ptr)

3. Shrink 64-bit integers to 32-bit integers
   - Allows each pool to be up to 4GB in size

Layout after pointer compression
Talk Outline

- Introduction & Motivation
- **Automatic Pool Allocation Background**
- Pointer Compression Transformation
- Experimental Results
- Conclusion
1. Compute points-to graph:
   - Ensure each pointer has one target
     - “unification-based” approach

2. Infer pool lifetimes:
   - Uses escape analysis

3. Rewrite program:
   - `malloc → poolalloc`, `free → poolfree`
   - Insert calls to `poolinit/pooldestroy`
   - Pass pool descriptors to functions

For more info: see MSP paper or talk at PLDI tomorrow

Chris Lattner
A Simple Pointer-intensive Example

A list of pointers to doubles

List *L = 0;
for (...) {
    List *N = malloc(List);
    N->Next = L;
    N->Data = malloc(double);
    L = N;
}

Pool allocate

List *L = 0;
for (...) {
    List *N = poolalloc(PD1, List);
    N->Next = L;
    N->Data = poolalloc(PD2, double);
    L = N;
}
Effect of Automatic Pool Allocation 1/2

- Heap is partitioned into separate pools
  - Each individual pool is smaller than total heap

```
List *L = 0;
for (...) {
    List *N = malloc(List);
    N->Next = L;
    N->Data = malloc(double);
    L = N;
}
```

```c
List *L = 0;
for (...) {
    List *N = poolalloc(PD1, List);
    N->Next = L;
    N->Data = poolalloc(PD2, double);
    L = N;
}
```

Chris Lattner
Each pool has a descriptor associated with it:
- Passed into poolalloc/poolfree

```c
List *L = 0;
for (...) {
    List *N = malloc(List);
    N->Next = L;
    N->Data = malloc(double);
    L = N;
}
```

We know which pool each pointer points into:
- Given the above, we also have the pool descriptor
- e.g. “N”, “L” → PD1 and N->Data → PD2
Talk Outline

- Introduction & Motivation
- Automatic Pool Allocation Background
- Pointer Compression Transformation
- Experimental Results
- Conclusion
Index Conversion of a Pool

- **Force pool memory to be contiguous:**
  - Normal PoolAlloc runtime allocates memory in chunks
  - Two implementation strategies for this (see paper)

- **Change pointers into the pool to integer offsets/indexes from pool base:**
  - Replace “*p” with “*(PoolBase + P)”

A pool can be index converted if pointers into it only point to heap memory (no stack or global mem)
Index Compression of a Pointer

- Shrink indexes in type-homogenous pools
  - Shrink from 64-bits to 32-bits
- Replace structure definition & field accesses
  - Requires accurate type-info and type-safe accesses

```c
struct List {
    struct List *Next;
    int Data;
};

List *L = malloc(16);
```

Chris Lattner
Index Conversion Example 1

Previous Example

Two pointers are compressible

After Index Conversion

Index conversion changes pointer dereferences, but not memory layout

64-bit integer indexes
Index Compression Example 1

Example after index conversion

Compress both indexes from 64 to 32-bit ints

Compress pointers, change accesses to and size of structure

‘Pointer’ registers remain 64-bits
Impact of Type-Homogeneity/safety

Compression requires rewriting structures:
- e.g. malloc(32) → malloc(16)
- Rewriting depends on type-safe memory accesses
  - We can’t know how to rewrite unions and other cases
- Must verify that memory accesses are ‘type-safe’

Pool allocation infers type homogeneity:
- Unions, bad C pointer tricks, etc → non-TH
- Some pools may be TH, others not

Can’t index compress pointers in non-TH pools!
Index Conversion Example 2

Heap pointer points from TH pool to a heap-only pool: compress this pointer!

Index conversion changes pointer dereferences, but not memory layout.

Compression of TH memory, pointed to by non-TH memory.
Index Compression Example 2

Example after index conversion

Next step: compress the pointer in the heap

Compress pointer in type-safe pool, change offsets and size of structure

Compression of TH memory, pointed to by non-TH memory
Compression of TH pointers, pointing to non-TH memory

Index convert non-TH pool to shrink TH pointers

Index compress array of pointers!
Static Pointer Compression Impl.

- Inspect graph of pools provided by APA:
  - Find compressible pointers
  - Determine pools to index convert

- Use rewrite rules to ptr compress program:
  - e.g. if $P_1$ and $P_2$ are compressed pointers, change:
    
    $P_1 = \*P_2 \Rightarrow P_1' = \*(\text{int}*)(\text{PoolBase}+P_2')$

- Perform interprocedural call graph traversal:
  - Top-down traversal from main()
Talk Outline

- Introduction & Motivation
- Automatic Pool Allocation Background
- Pointer Compression Transformation
- Dynamic Pointer Compression
- Experimental Results
- Conclusion
Dynamic Pointer Compression Idea

Static compression can break programs:
- Each pool is limited to $2^{32}$ bytes of memory
  - Program aborts when $2^{32}$nd byte allocated!

Expand pointers dynamically when needed!
- When $2^{32}$nd byte is allocated, expand pointers to 64-bits
- Traverse/rewrite pool, uses type information
- Similar to (but a bit simpler than) a copying GC pass
Structure offset and sizes depend on whether a pool is currently compressed:

\[ P_1 = \ast P_2 \implies \text{if } (PD->\text{isCompressed}) \]

\[ P_1' = \ast(int32*)(PoolBase + P_2'\ast C1 + C2); \]
else

\[ P_1' = \ast(int64*)(PoolBase + P_2'\ast C3 + C4); \]

Use standard optimizations to address this:
- Predication, loop unswitching, jump threading, etc.

See paper for details.
Talk Outline

- Introduction & Motivation
- Automatic Pool Allocation Background
- Pointer Compression Transformation
- Experimental Results
- Conclusion
Experimental Results: 2 Questions

1. Does Pointer Compression improve the performance of pointer intensive programs?
   - Cache miss reductions, memory bandwidth improvements
   - Memory footprint reduction

2. How does the impact of Pointer Compression vary across 64-bit architectures?
   - Do memory system improvements outweigh overhead?

Built in the LLVM Compiler Infrastructure: http://llvm.cs.uiuc.edu/
Chris Lattner
Static PtrComp Performance Impact

1.0 = Program compiled with LLVM & PA but no PC

Peak Memory Usage

<table>
<thead>
<tr>
<th></th>
<th>PA</th>
<th>PC</th>
<th>PC/PA</th>
</tr>
</thead>
<tbody>
<tr>
<td>bh</td>
<td>8MB</td>
<td>8MB</td>
<td>1.00</td>
</tr>
<tr>
<td>bisort</td>
<td>64MB</td>
<td>32MB</td>
<td>0.50</td>
</tr>
<tr>
<td>em3d</td>
<td>47MB</td>
<td>47MB</td>
<td>1.00</td>
</tr>
<tr>
<td>perimeter</td>
<td>299MB</td>
<td>171MB</td>
<td>0.57</td>
</tr>
<tr>
<td>power</td>
<td>882KB</td>
<td>816KB</td>
<td>0.93</td>
</tr>
<tr>
<td>treeadd</td>
<td>128MB</td>
<td>64MB</td>
<td>0.50</td>
</tr>
<tr>
<td>tsp</td>
<td>128MB</td>
<td>96MB</td>
<td>0.75</td>
</tr>
<tr>
<td>ft</td>
<td>9MB</td>
<td>4MB</td>
<td>0.51</td>
</tr>
<tr>
<td>ks</td>
<td>47KB</td>
<td>47KB</td>
<td>1.00</td>
</tr>
<tr>
<td>llubench</td>
<td>4MB</td>
<td>2MB</td>
<td>0.50</td>
</tr>
</tbody>
</table>

UltraSPARC IIIi w/1MB Cache

Chris Lattner
Evaluating Effect of Architecture

- Pick one program that scales easily:
  - llubench – Linked list micro-benchmark
  - llubench has little computation, many dereferences
    - Best possible case for pointer compression
- Evaluate how ptrcomp impacts scalability:
  - Compare to native and pool allocated version
- Evaluate overhead introduced by ptrcomp:
  - Compare PA32 with PC32 (‘compress’ 32 → 32 bits)
- How close is ptrcomp to native 32-bit pointers?
  - Compare to native-32 and poolalloc-32 for limit study
Chris Lattner

AMD64 PtrComp (1MB Cache)

Heap Size (no units)

Time per node dereference

Similar scalability impact

PtrComp is faster than PA32 on AMD64!

See paper for IA64 and IBM-SP

Chris Lattner
Pointer compression can substantially reduce footprint of pointer-intensive programs:
- … without specialized hardware support!

Significant perf. impact for some programs:
- Effectively higher memory bandwidth
- Effectively larger caches

Dynamic compression for full generality:
- Speculate that pools are small, expand if not
- More investigation needed, see paper!

Questions?

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