Making Context-sensitive Points-to Analysis with Heap Cloning Practical For The Real World

Chris Lattner
Apple

Andrew Lenharth
UIUC

Vikram Adve
UIUC

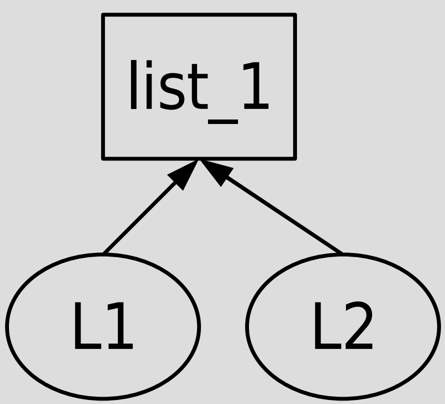
What is Heap Cloning?

Distinguish objects by acyclic call path

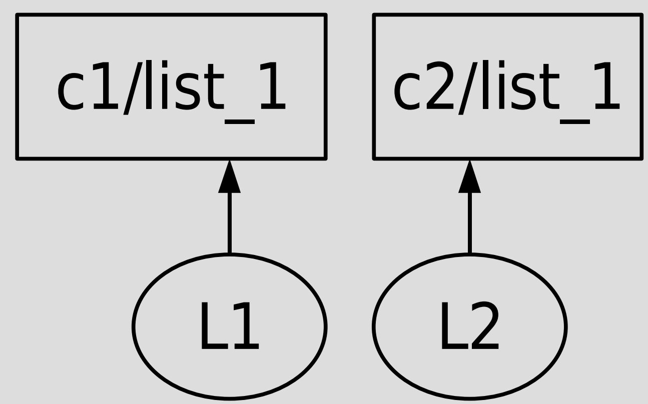
```
void foo() {
c1:  list* L1 = mkList(10);
c2:  list* L2 = mkList(10);
}
```

```
list* mkList(int num) {
  list* L = NULL;
  while (--num)
list_1:  L = new list(L);
}
```

Without heap cloning:
Lists are allocated in a common place so they are the same list



With heap cloning:
Disjoint data structure instances are discovered



Why Heap Cloning?

- Discover disjoint data structure instances
  - able to process and/or optimize each instance
- More precise alias analysis
- Important in discovering coarse grain parallelism*
- More precise shape analysis?

But widely considered
non-scalable and rarely used

* Ryoo et. al., HiPEAC '06

Some Uses of Our Analysis

Data Structure Analysis (DSA) is well tested, used for major program transformations

- Automatic Pool Allocation
  - PLDI 2005 – Best Paper
- Pointer Compression
  - MSP 2005
- SAFECode
  - PLDI 2006
- Less conservative GC
- Per-instance profiling
- Alias Analysis
  - optimizations that use alias results

Available at llvm.org

Key Contributions

Heap cloning (with unification)
can be scalable and fast

- Many algorithmic choices, optimizations necessary
  - We **measure** several of them
- Sound and useful analysis on incomplete programs
- New techniques
  - Fine-grained completeness tracking solves 3 practical issues
  - Call graph discovery during analysis, no iteration
  - New engineering optimizations

Outline

- Algorithm overview
- Results summary
- Optimizations and their effectiveness

Design Decisions

Fast analysis and scalable for production compilers!

Common Design of Scalable Algorithms

Improves Speed, Hurts Precision

- Unification based
- Flow insensitive
- Drop context-sensitivity in SCCs of call graph

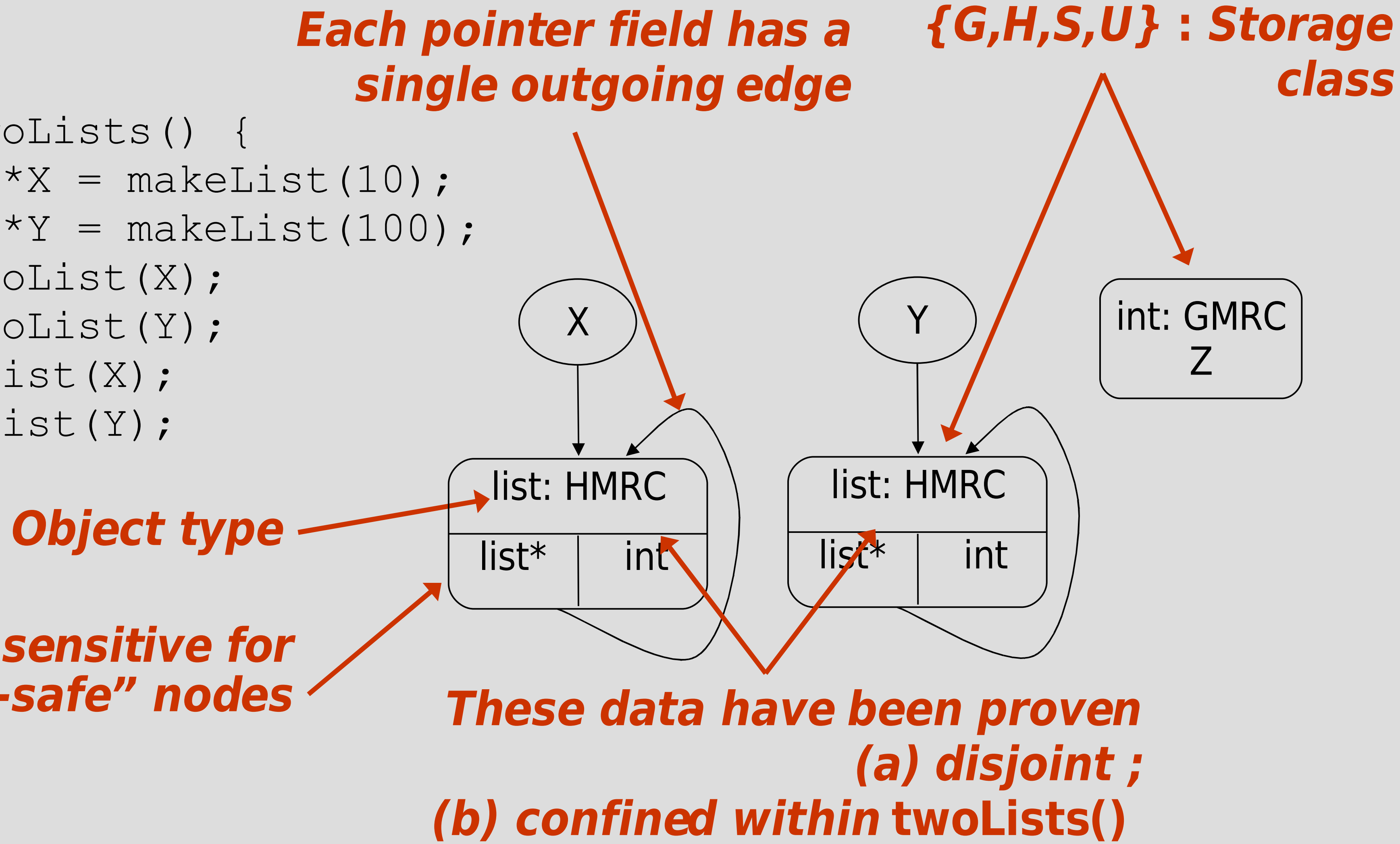
Improves Precision

- Field sensitive
- Context sensitive
- Heap cloning
- Fine-grained completeness
- Use-based type inferencing for C

DS Graph Properties

```
int Z;

void twoLists() {
  list *X = makeList(10);
  list *Y = makeList(100);
  addGToList(X);
  addGToList(Y);
  freeList(X);
  freeList(Y);
}
```



Algorithm Fly-by

3 Phase Algorithm

- Local
  - Field-sensitive intra-procedural summary graph
- Bottom-up on SCCs of the call graph
  - Clone and inline callees into callers
  - summary of full effects of calling the function
- Top-down on SCCs of the call graph
  - Clone and inline callers into callees

Completeness

A graph node is complete if we can prove we have seen all operations on its objects

1. Support incomplete programs
2. Safely speculate on type safety
3. Construct call graph incrementally

Incompleteness - Sources

Incompleteness is a transitive closure starting from escaping memory:

```
list* ExternGV;
static int LocalGV;

int* escaping_fun(list*) {...}

static int* local_fun(list*) {
...
x = extern_fun(L1);
...
}
```

Externally visible globals

Return values and arguments of escaping functions

Return value and arguments of external or unresolved indirect calls

Call Graph Discovery

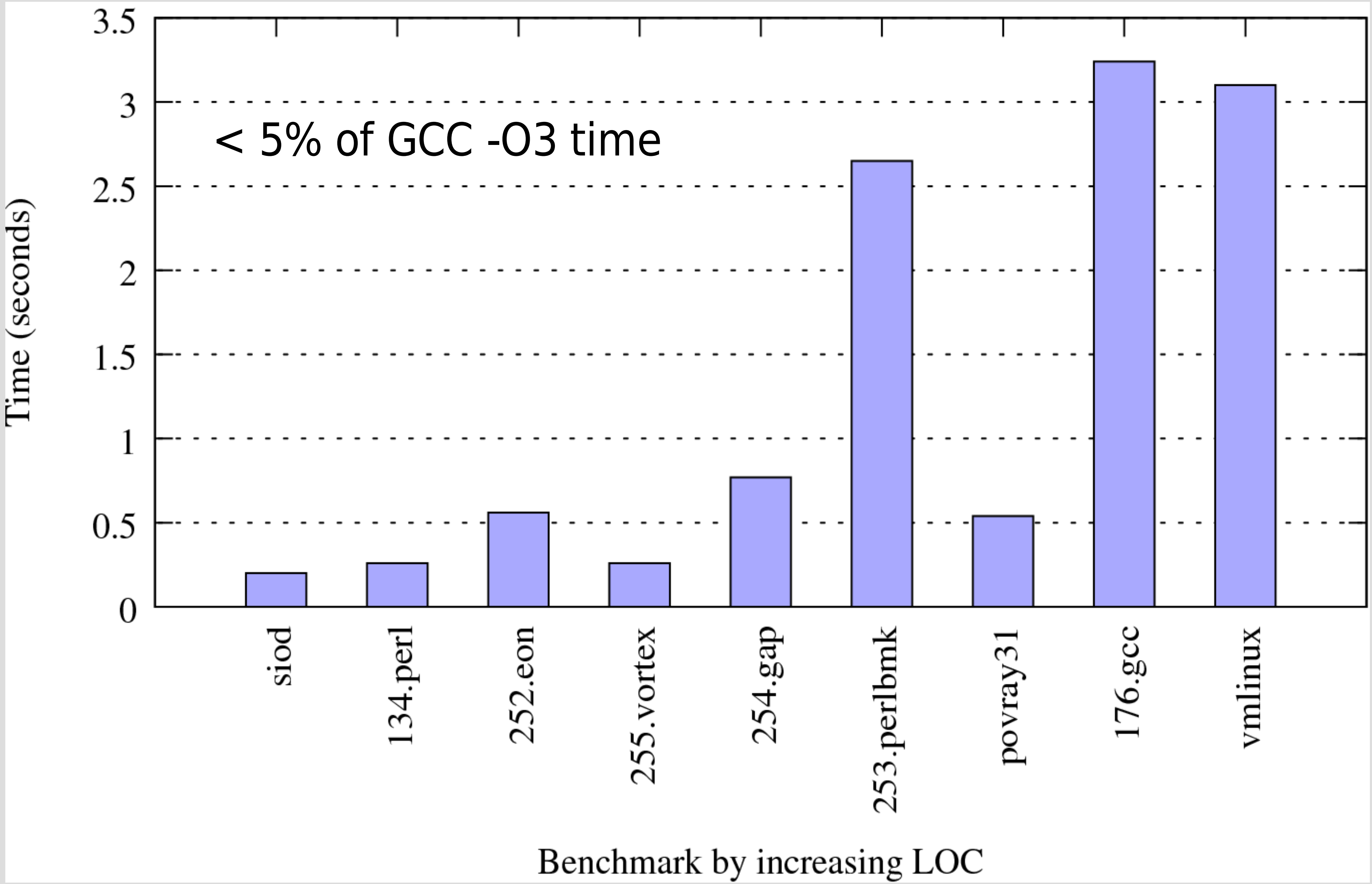
- Discover call targets in a context-sensitive way
- Incompleteness ensures correctness of points-to graphs with unresolved call sites
- SCCs may be formed by resolving an indirect call
  - Key insight: safe to process SCC even if some of its functions are already processed
  - See paper for details

Methodology

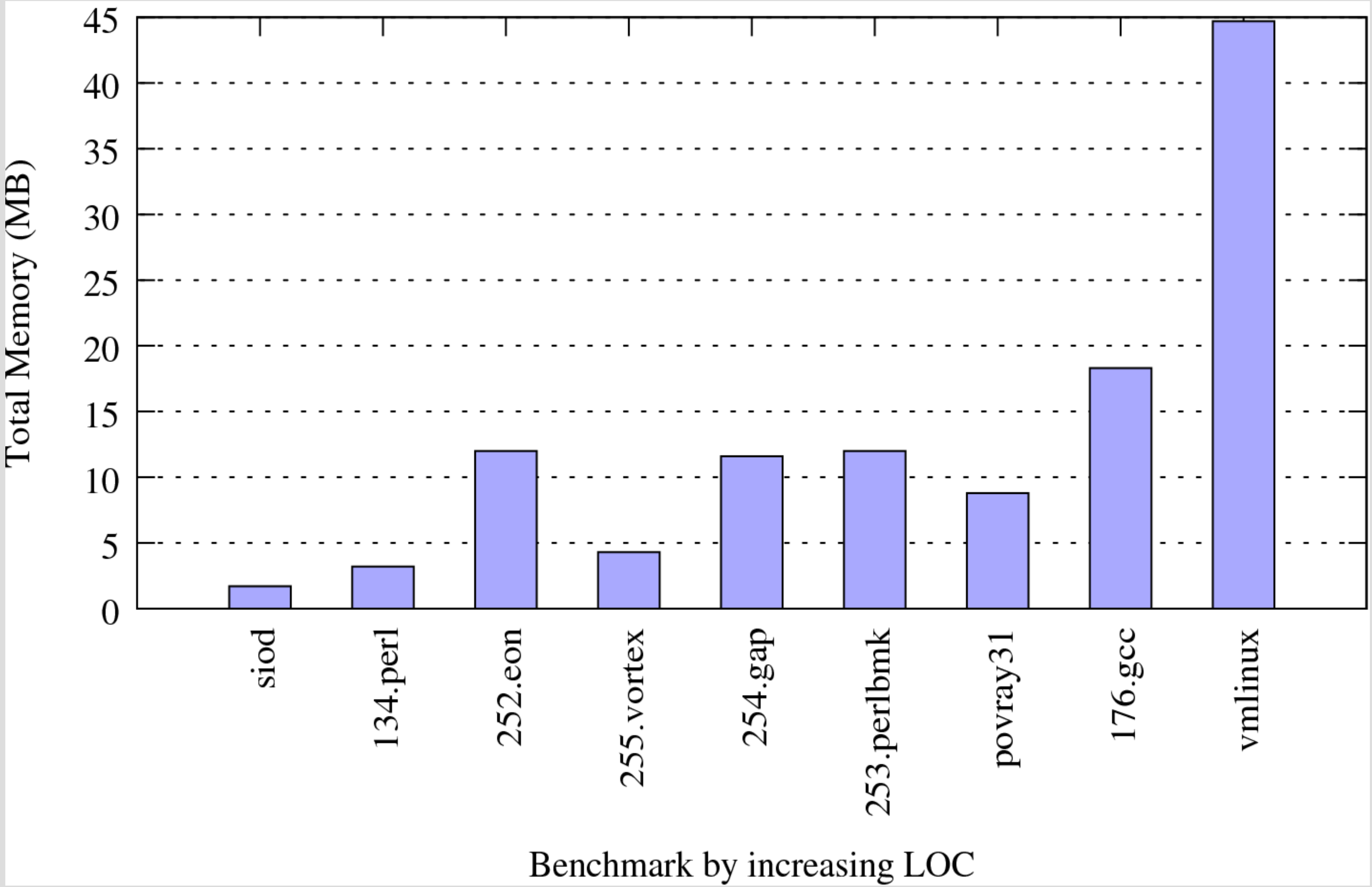
- Benchmarks:
  - SPEC 95 and 2000
  - Linux 2.4.22
  - povray 3.1
  - Ptrdist
- Presenting 9 benchmarks with slowest analysis time
  - Except 147.vortex and 126.gcc
  - Lots more in paper
- Machine: 1.7 Ghz AMD Athlon, 1 GB Ram

Benchmark	**kLOC**
siod	12.8
134.perl	26.9
252.eon	35.8
255.vortex	67.2
254.gap	71.3
253.perlbmk	85.1
povray31	108.3
176.gcc	222.2
vmlinux	355.4

Results - Speed



Results – Memory Usage

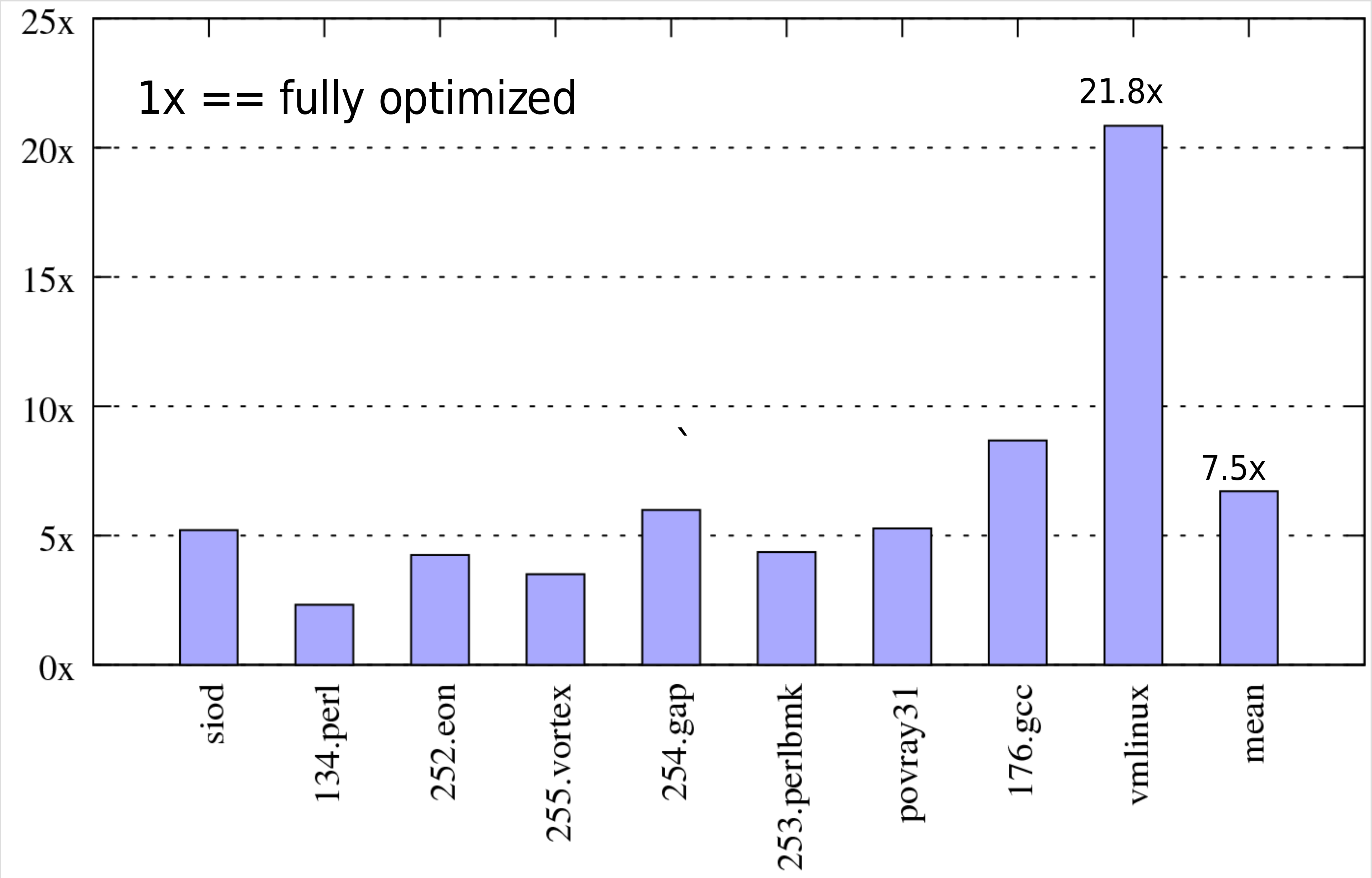


Avoiding Bad Behavior

- Equivalence classes
  - Avoid N^2 space and time for globals not used in most functions
- Globals Graph*
  - Avoid N^2 replication of globals in nodes
- SCC collapsing*
  - Avoid recursive inlining
  - hurts precision
- Optimized Cloning and Merging*
  - Avoid lots of allocation traffic

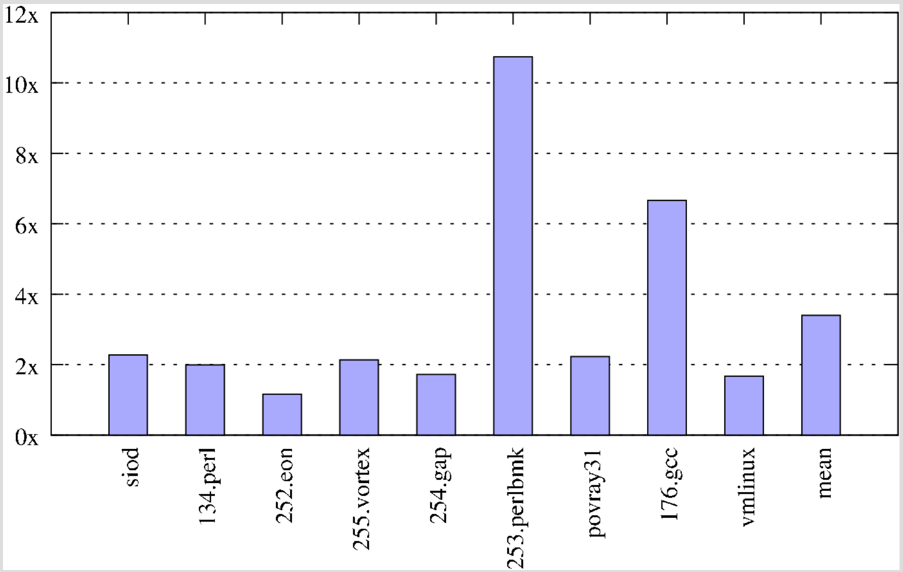
* used by others also

Slowdowns – No Optimizations

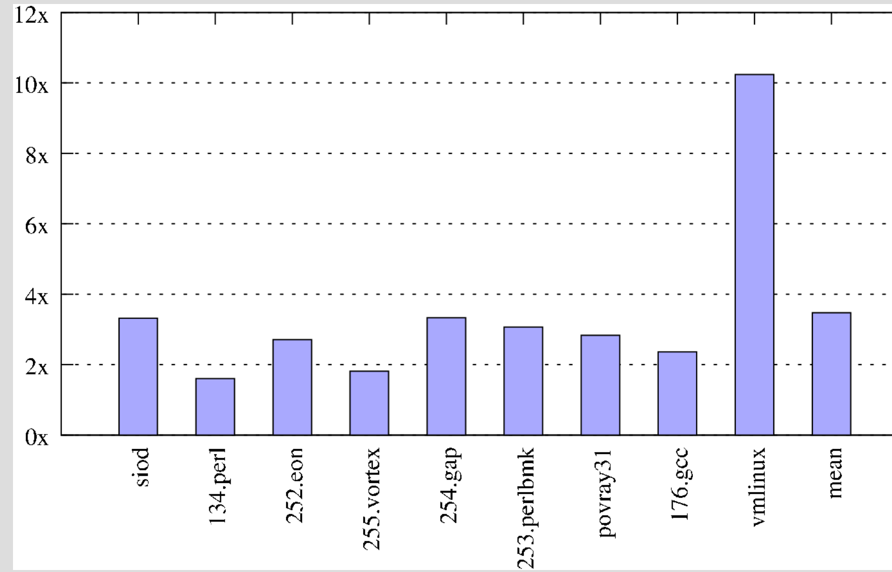


Optimizations Effects

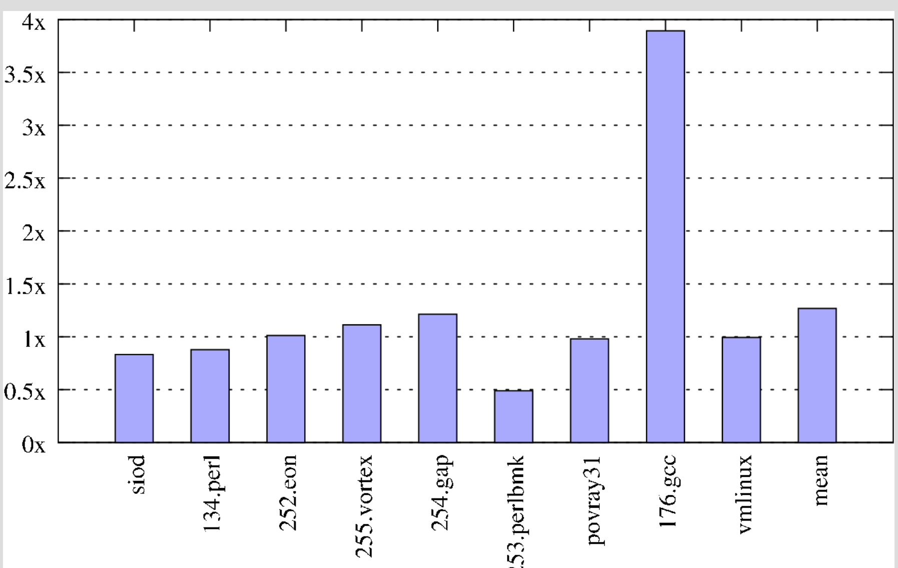
Naive Merging



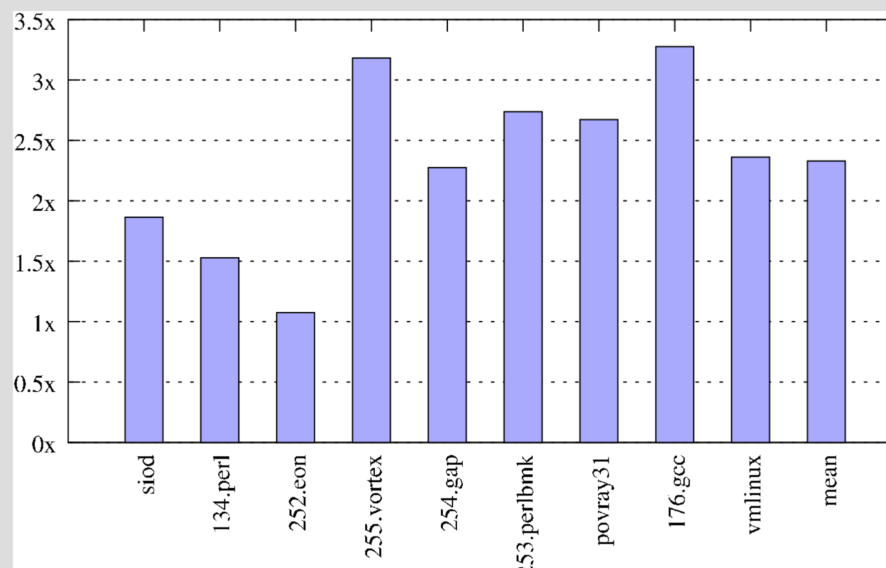
No Equivalence Classes



No SCC Collapsing



No Globals Graph



Results – By Size

Speedup due to optimizations grows as program size does

	Average LOC	Average Speedup
Largest 4 programs	280k	10.8x
Second largest 4	72k	4.4x
Third largest 4	52k	2.7x

Optimizations are essential for scalability, not just speed

Summary

- Context sensitive analyses with heap cloning can be efficient enough for production compilers
- Sound and useful analysis is possible on incomplete programs
- Many optimizations necessary for speed and scalability

Questions?

Rob: Why heap cloning?
Andrew: It's better than sheep cloning.
Rob: Yes, heap cloning raises none of the ethical concerns of sheep cloning, and sometimes the sheep have strange developmental issues that you don't get with heap cloning.

Related – Ruf

Similarities

- Unification
- Heap cloning
- Field sensitive
- Globals graph
- Intelligent inlining
- Drop context sensitivity in SCC

Differences

- Requires whole program
- For type safe language
- Requires call graph
  - used context insensitive

Related – Liang (FICS)

Similarities

- Unification
- Context sensitive
- Field sensitive

Differences

- Iterates during Bottom Up
- No heap cloning
- Requires call graph

Related – Liang (MOPPA)

Similarities

- Unification
- Context sensitive
- Field sensitive
- Globals graph
- Heap Cloning

Differences

- Iterates during Bottom Up
- Requires call graph or iterates to construct it
- Memory intensive

Related - Whaley-Lam

Similarities

- Context sensitive

Differences

- Constraint solving algorithm
- Call graph is input to context-sensitive alg
  - discovered by context-insensitive alg
- For type safe language
- No heap cloning
- Much slower on similar hardware

Related - Bodik

Similarities

- Context sensitive
- Heap cloning
- SCC collapsing

Differences

- Subset based
- Requires call graph
- Demand driven
- Requires whole program
- For type safe language
- Much slower on similar hardware

Related - Nystron

- Top-down, bottom-up structure
- Context sensitive
- Heap cloning
- SCC collapsing
- Behavior of Globals stored in side structure
- Subset based
- Some codes cause runtime explosion

Why Heap Cloning? Part 2!

- Rob: Why heap cloning?
- Andrew: It's better than sheep cloning.
- Rob: Yes, heap cloning raises none of the ethical concerns of sheep cloning, and sometimes the sheep have strange developmental issues that you don't get with heap cloning.