# 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**

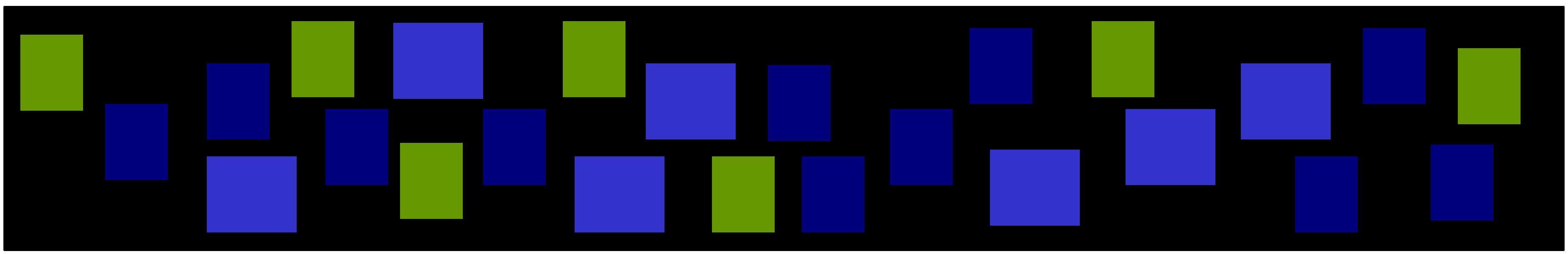
**PLDI 2005**

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

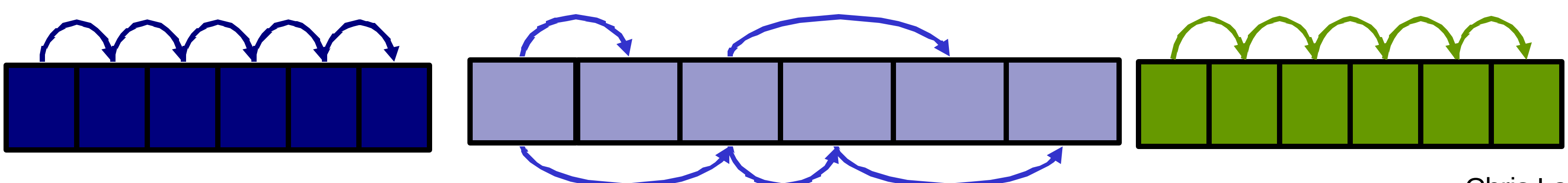
# What is the problem?


**What the program creates:**

**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

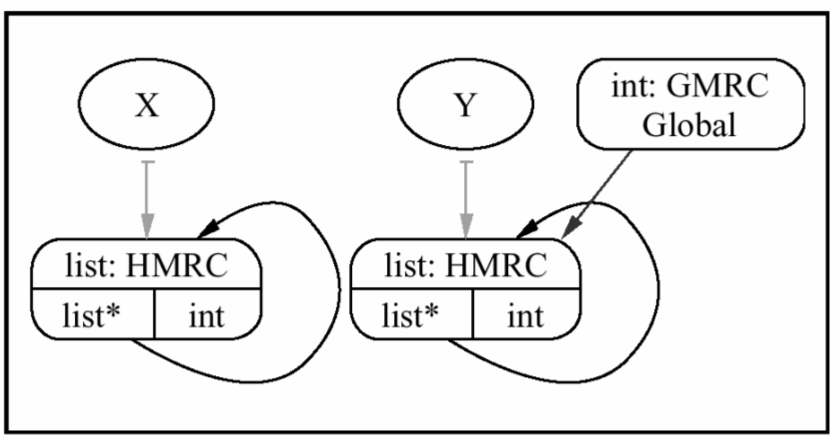
# Talk Outline

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

# Automatic Pool Allocation Overview

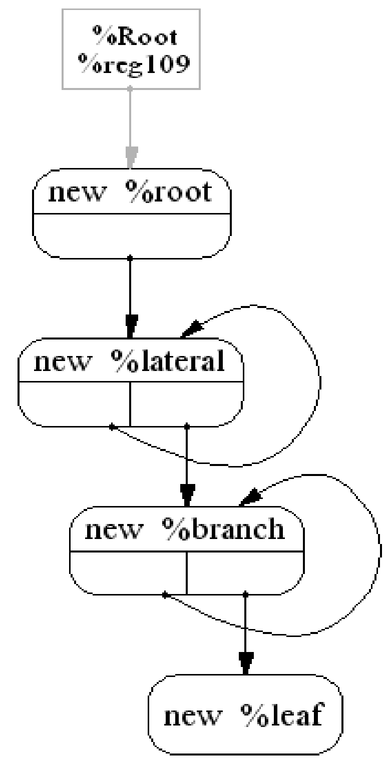
- **Segregate memory according to points-to graph**
- **Use context-sensitive analysis to distinguish between RDS instances passed to common routines**

**Points-to graph (two disjoint linked lists)**

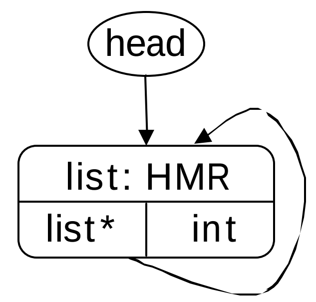



*Pool 1* *Pool 2*

*Pool 1*
*Pool 2*
*Pool 3*
*Pool 4*



# Points-to Graph Assumptions



Linked List

- **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

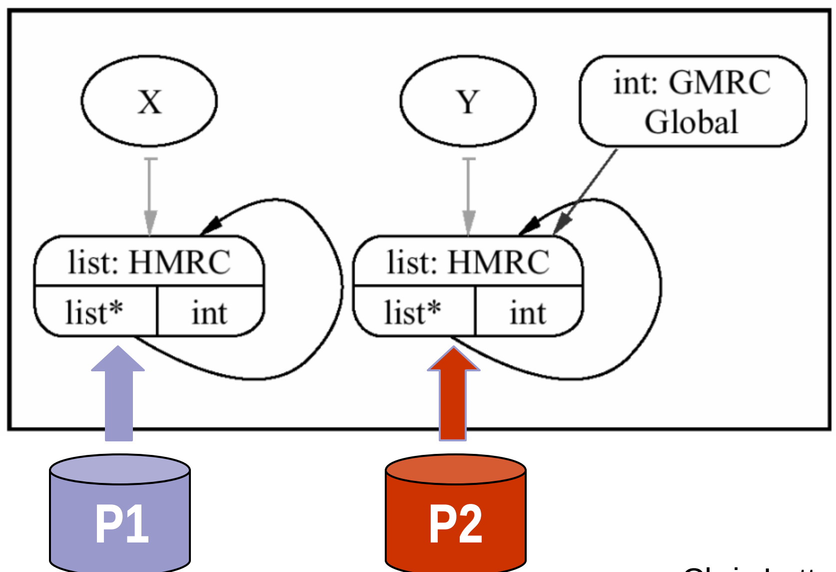
# Pool Allocation: Example

```
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(10, P2)
  GL = Y;
  processList(X);
  processList(Y);
  freeList(X, &P1)
  freeList(Y, P2)
 pooldestroy(&P1);
}
```

Change calls to free into calls to poolfree → retain explicit deallocation



# 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

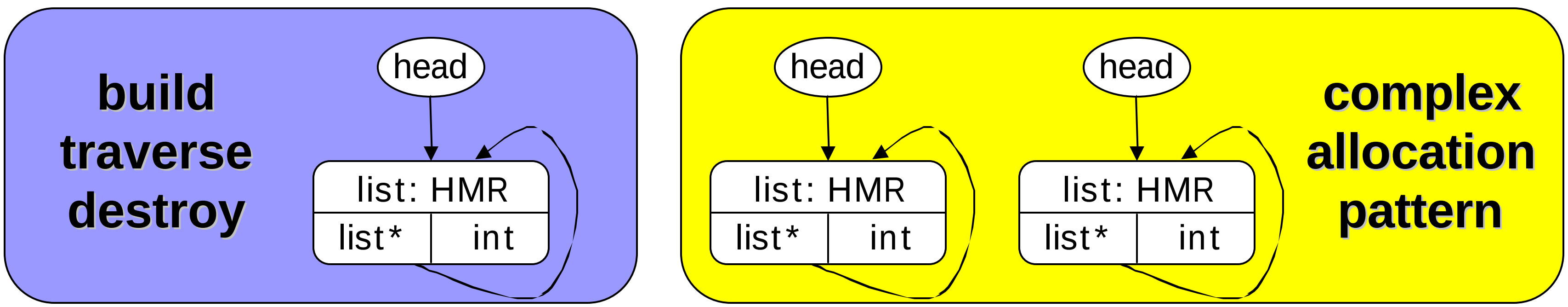
# Talk Outline

- 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***

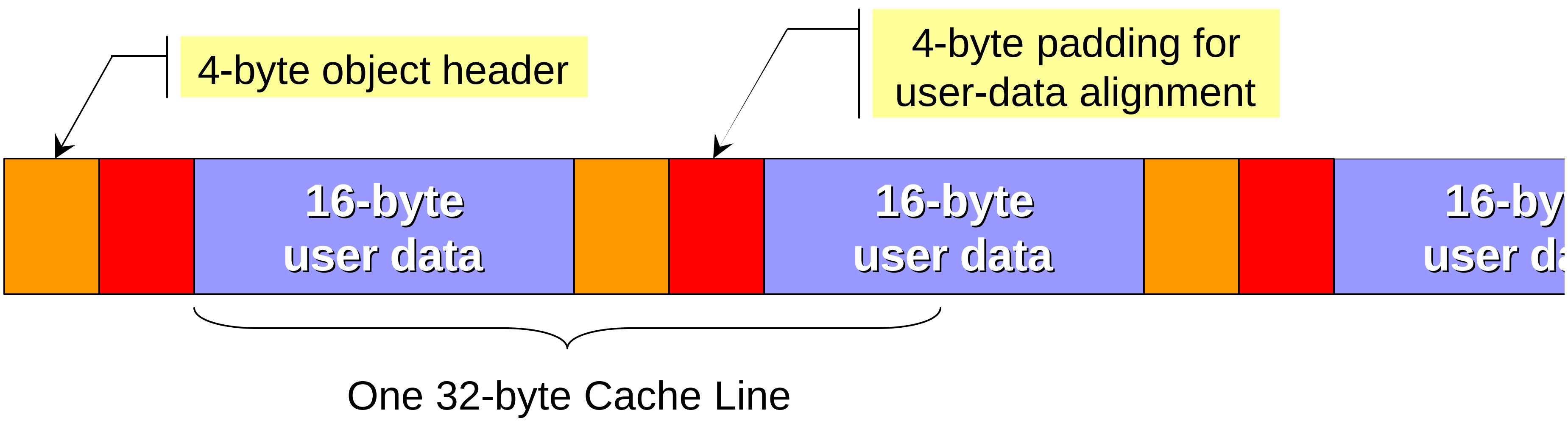
- **Pool allocation segregates heap:**
  - Roughly into logical data structures
  - Optimize using pool-specific properties




- **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):**




Chris Lattner

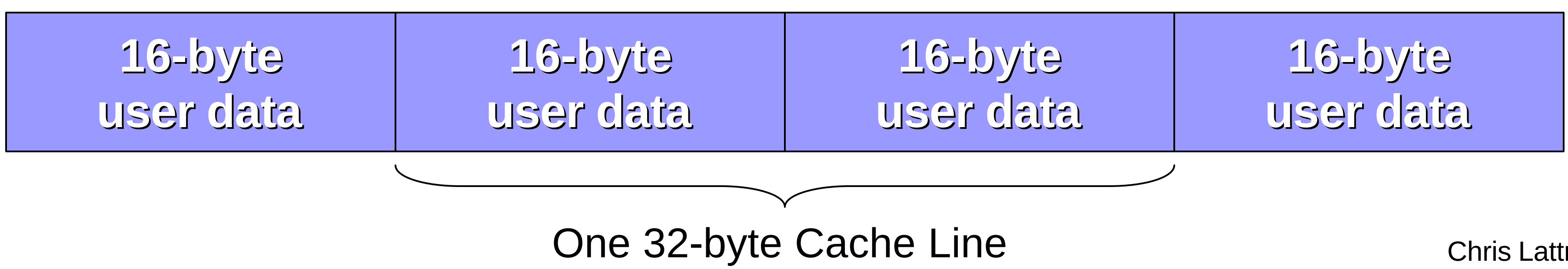
# PAOpts (1/4) and (2/4)

- **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!**

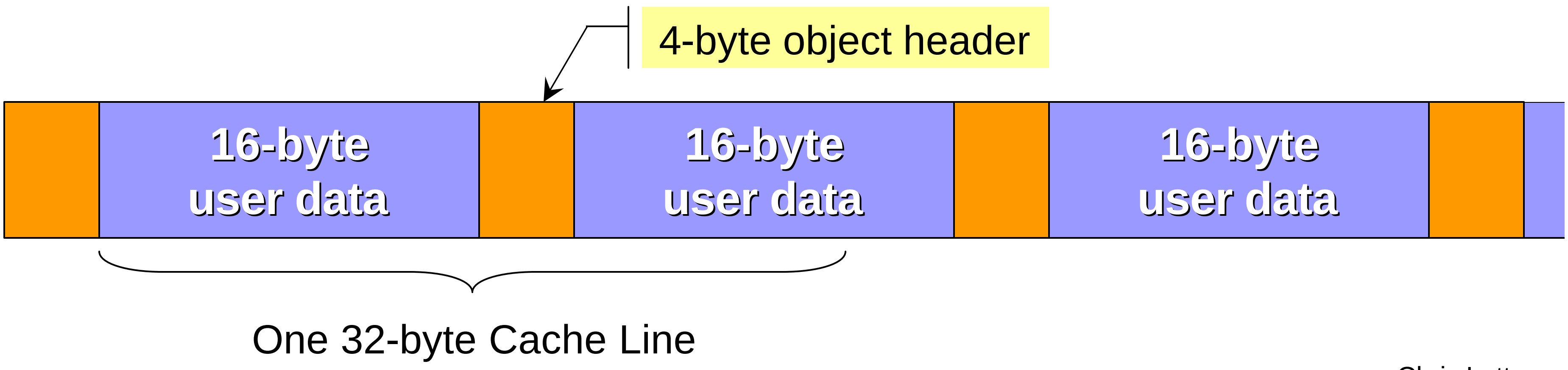
# PAOpts (3/4): Bump Pointer Optzn

- **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



# 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



# Talk Outline

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

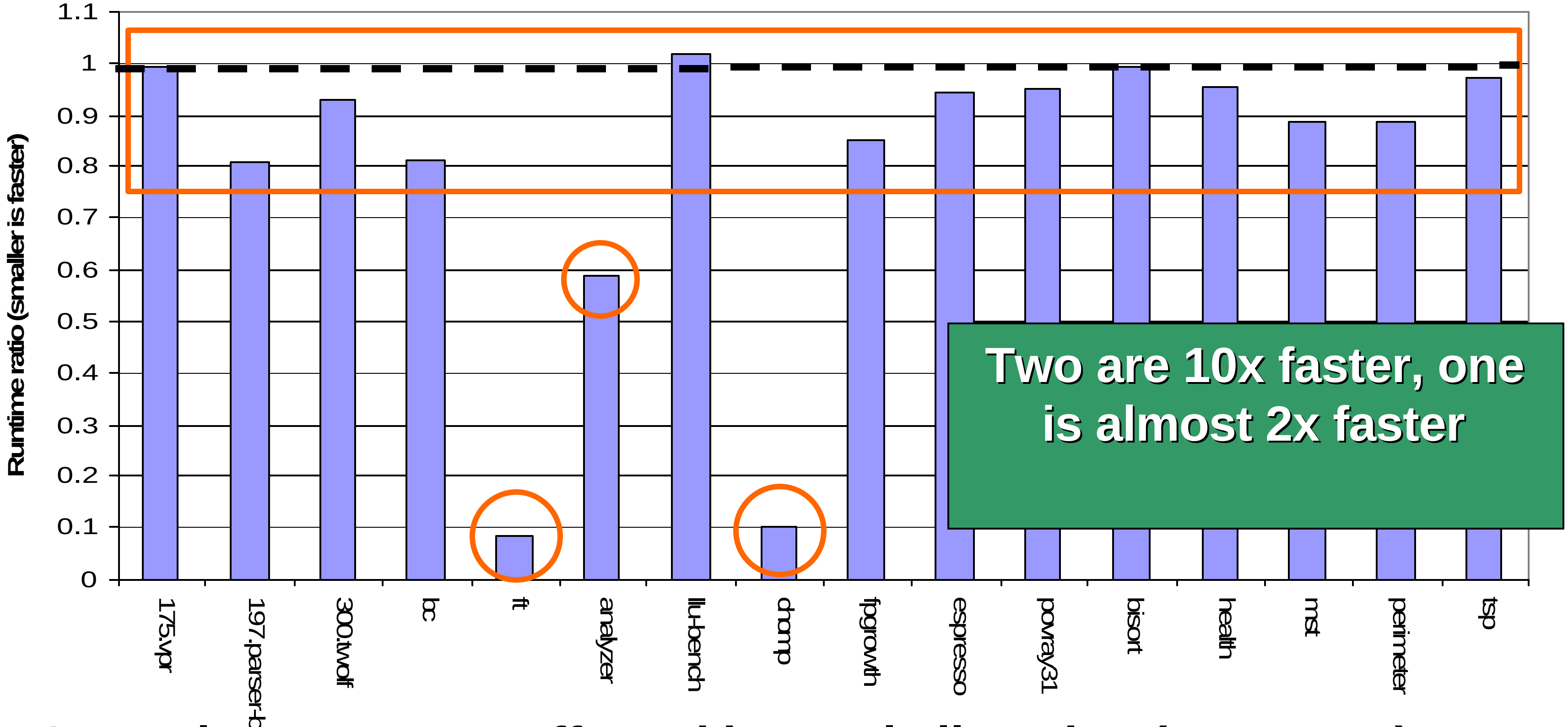
# Simple Pool Allocation Statistics

DSA is able to infer that most static pools are type-homogenous

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

DSA + Pool allocation compile time is small: less than 3% of GCC compile time for all tested programs. See paper for details

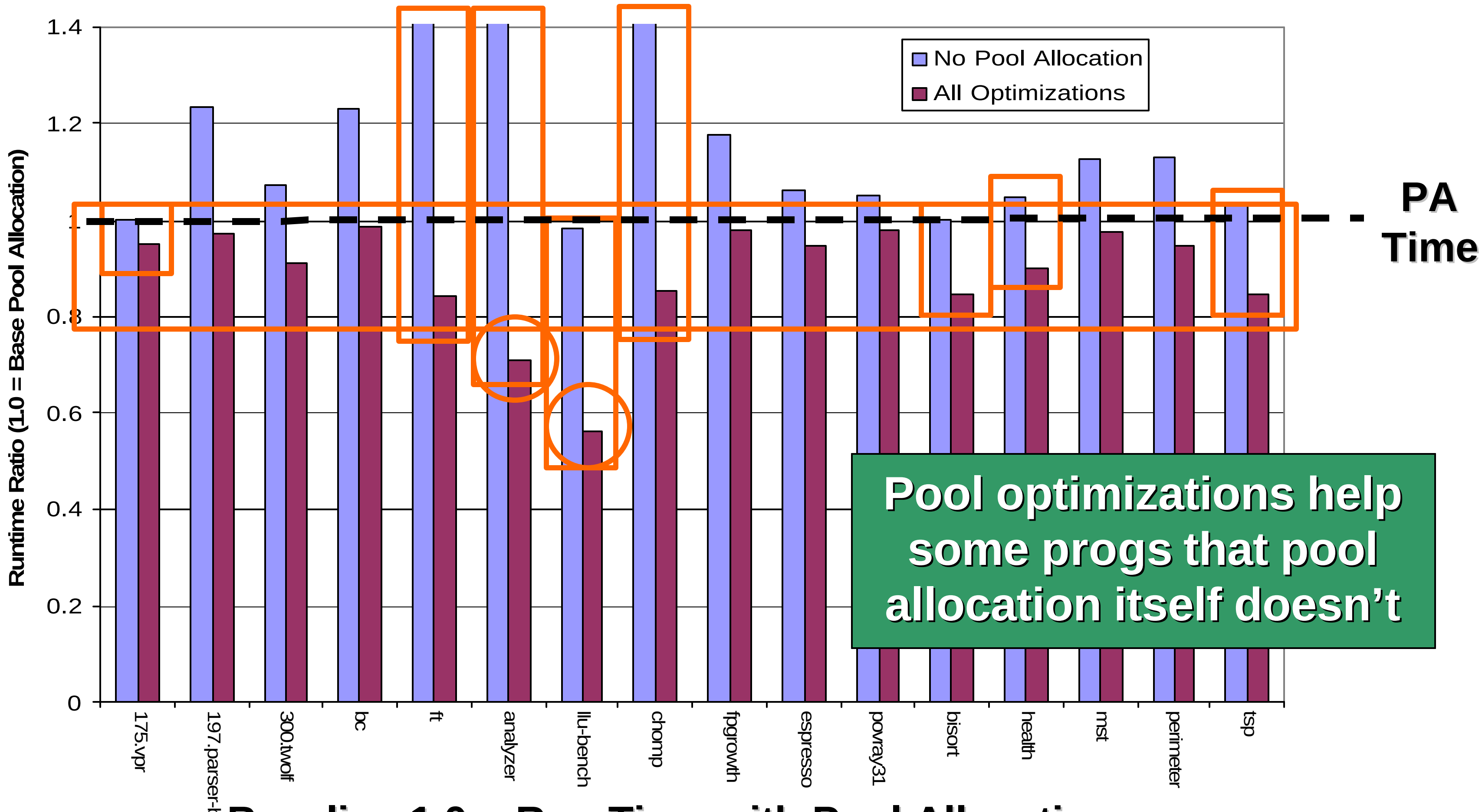
# Pool Allocation Speedup




- **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 Optimization Speedup (FullPA)

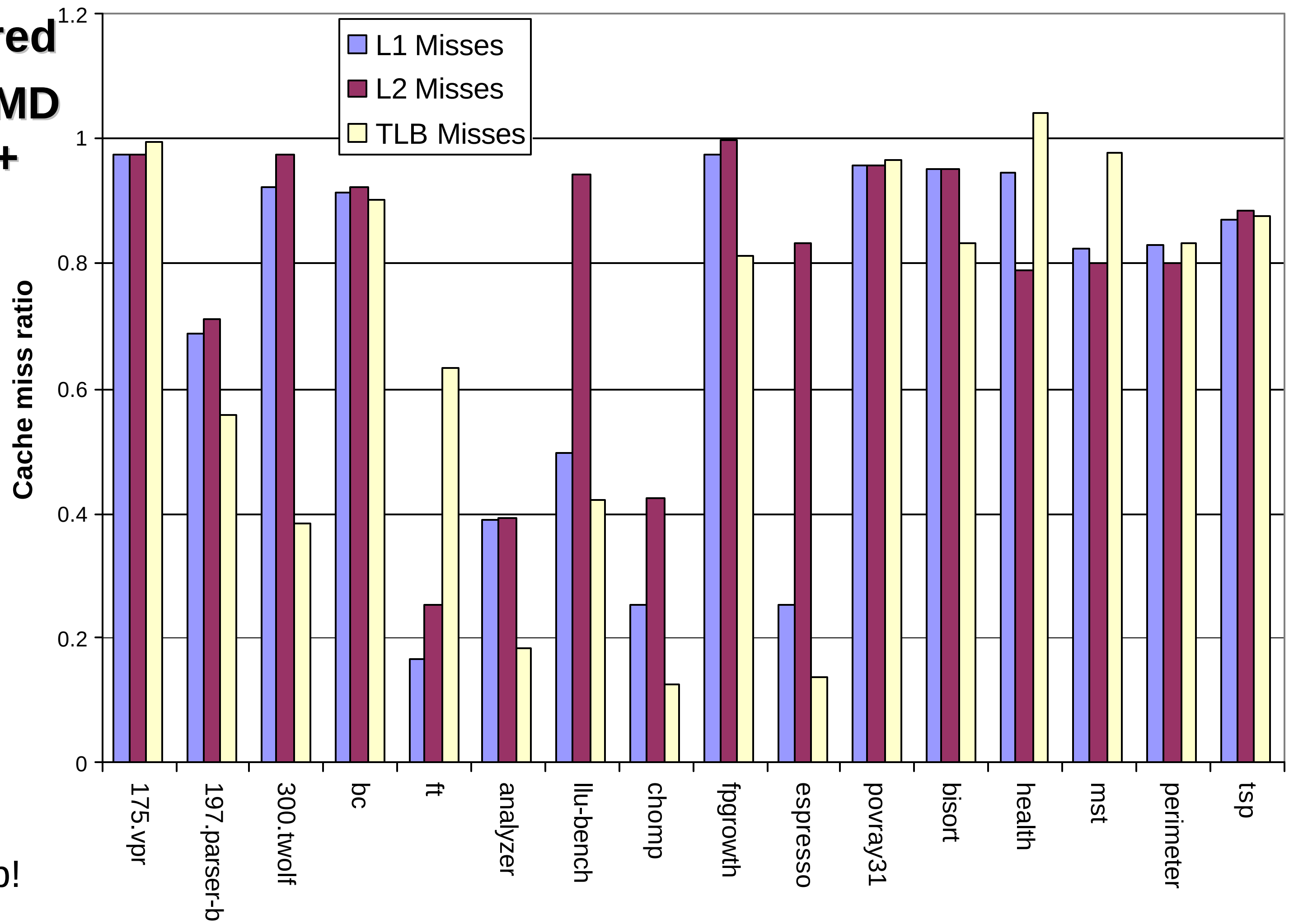



**Baseline 1.0 = Run Time with Pool Allocation**

- **Optimizations help all of these programs:**
  - Despite being very simple, they make a big impact

# Cache/TLB miss reduction

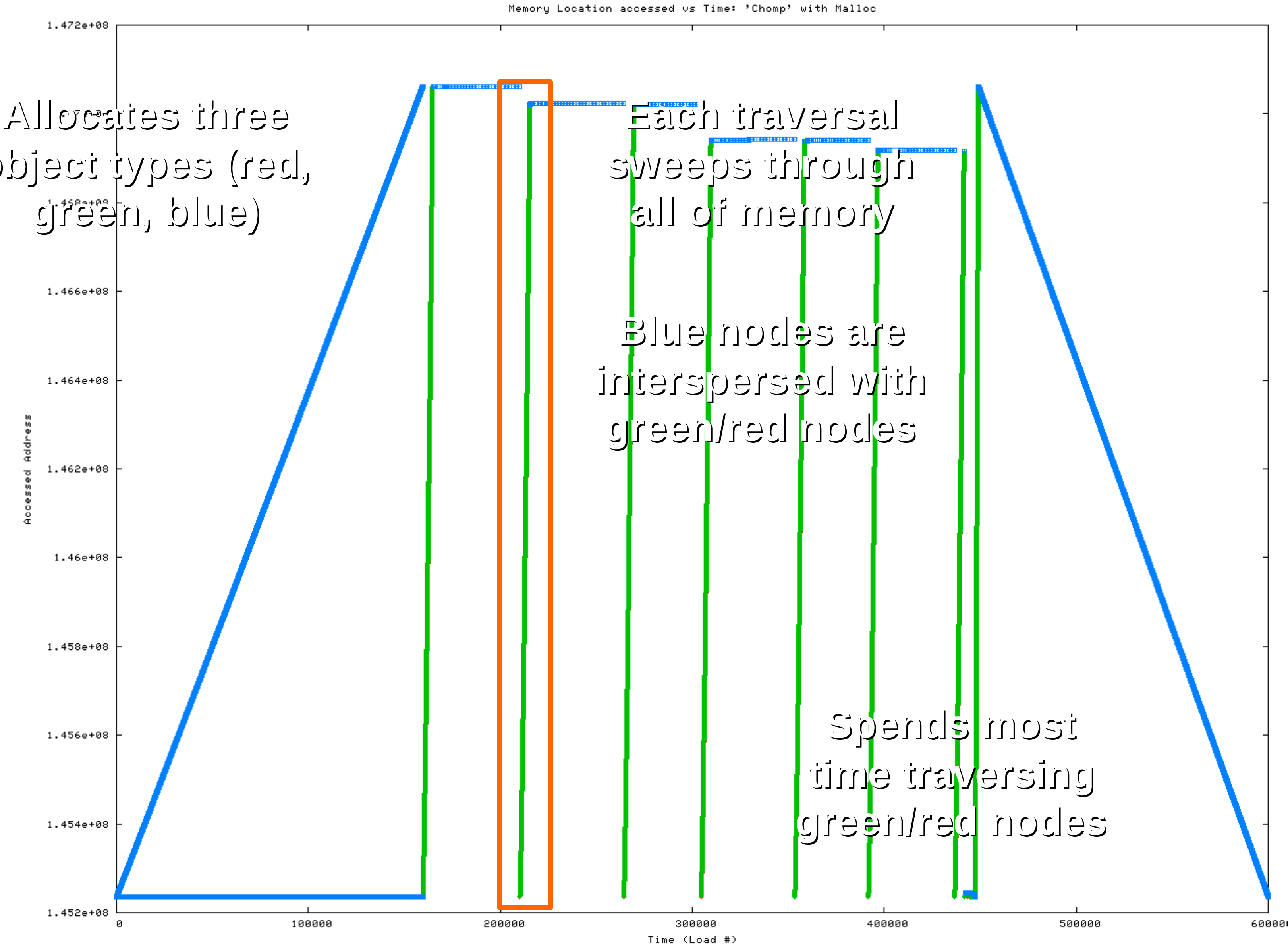
**Miss rate measured with perfctr on AMD Athlon 2100+**



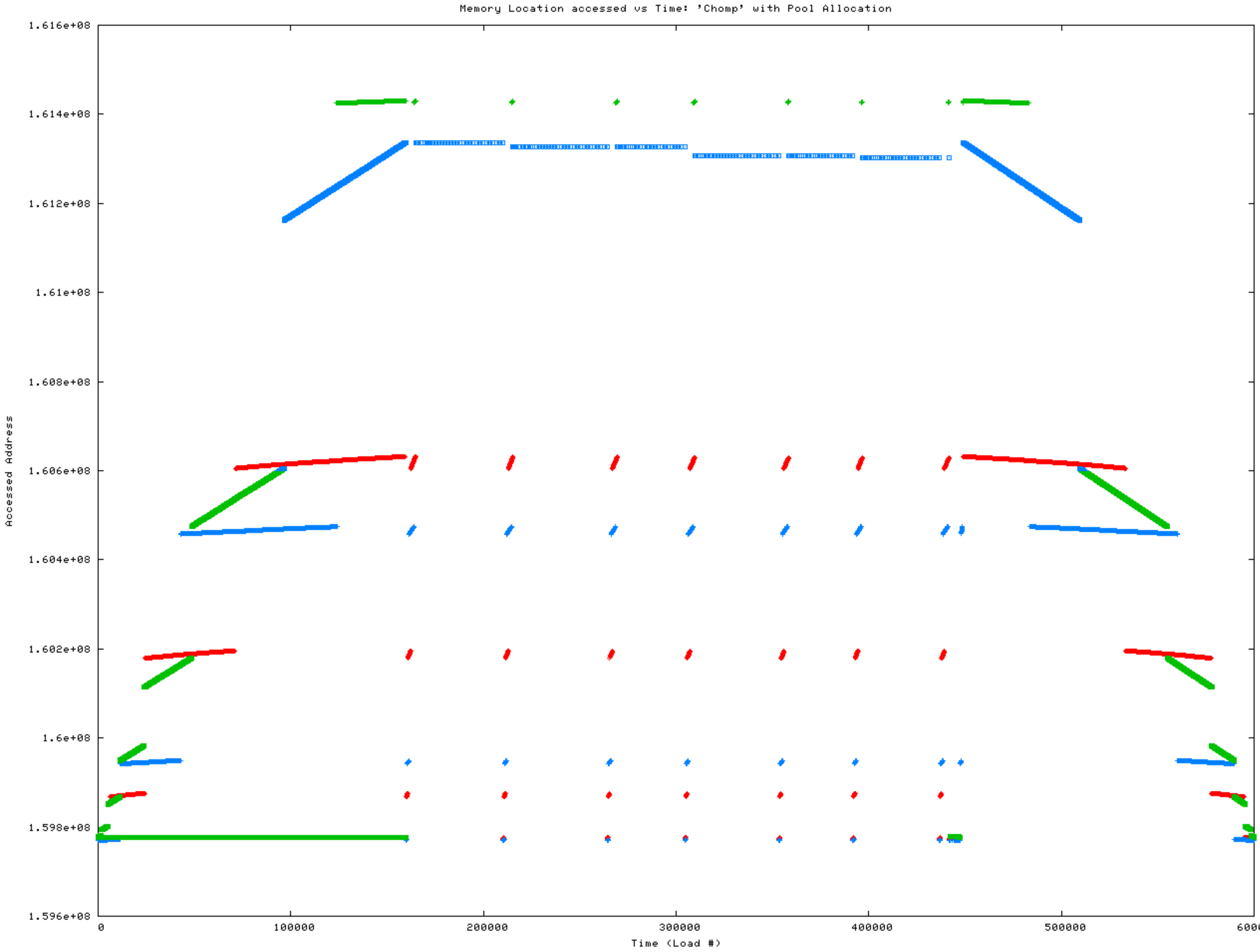

## Sources:

- Defragmented heap
- Reduced inter-object padding
- Segregating the heap!

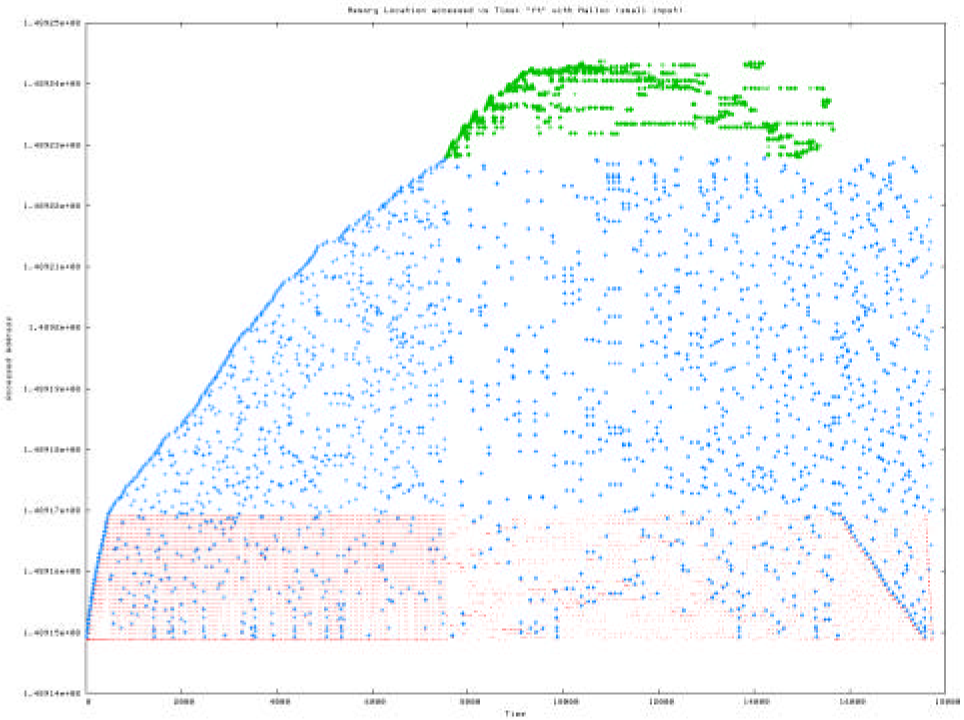
# Chomp Access Pattern with Malloc

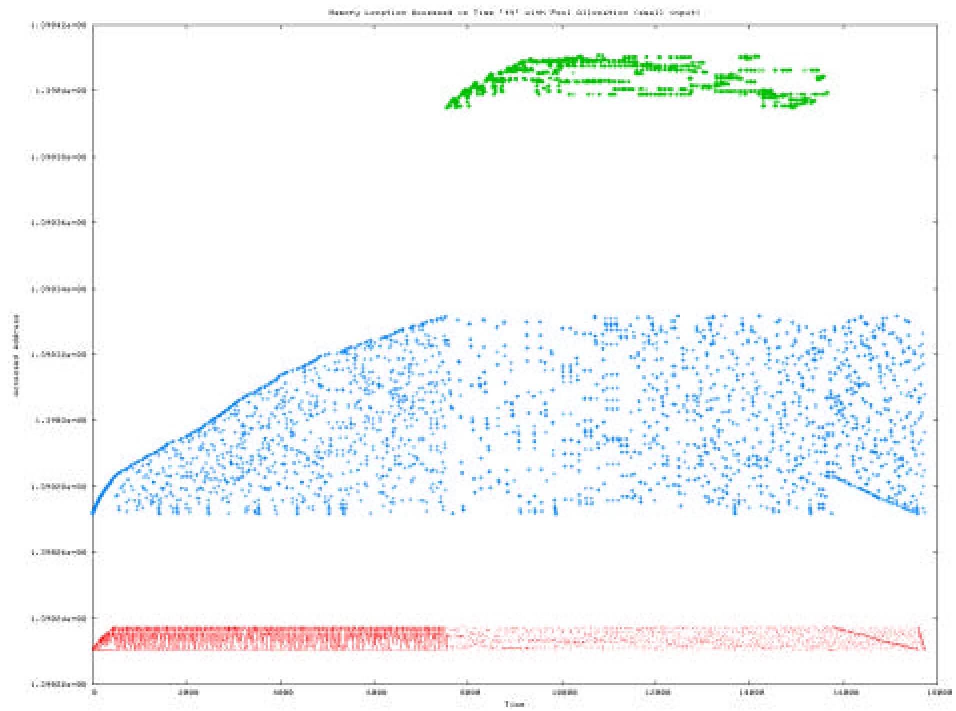


# Chomp Access Pattern with PoolAlloc



# FT Access Pattern With Malloc





- **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

# 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 ⇒ 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/