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

Chris Lattner Vikram Adve
University of Illinois at Urbana-Champaign
{lattner,vadve}@cs.uiuc.edu

Abstract

This paper describes *Automatic Pool Allocation*, a transformation framework that segregates distinct instances of heap-based data structures into seperate memory pools and allows heuristics to be used to partially control the internal layout of those data structures. The primary goal of this work is performance improvement, not automatic memory management, and the paper makes several new contributions. The key contribution is a new compiler algorithm for partitioning heap objects in imperative programs based on a context-sensitive pointer analysis, including a novel strategy for correct handling of indirect (and potentially unsafe) function calls. The transformation does not require type safe programs and works for the full generality of C and C++. Second, the paper describes several optimizations that exploit data structure partitioning to further improve program performance. Third, the paper evaluates how memory hierarchy behavior and overall program performance are impacted by the new transformations. Using a number of benchmarks and a few applications, we find that compilation times are extremely low, and overall running times for heap intensive programs speed up by 10-25% in many cases, about 2x in two cases, and more than 10x in two small benchmarks. Overall, we believe this work provides a new framework for optimizing pointer intensive programs by segregating and controlling the layout of heap-based data structures.

Categories and Subject Descriptors D.3.4 [*Processors*]: Compilers, Optimization, Memory management

General Terms Algorithms, Performance

Keywords Recursive data structure, data layout, cache, static analysis, pool allocation

1. Introduction

One of the most important tasks for modern compilers and runtime systems is the management of memory usage in programs, including safety checking, optimization, and storage management. Unfortunately, compilers have proved much more effective at analyzing and controlling memory access patterns for dense arrays than for pointer-based data structures. A key difference between the two is that compilers have precise knowledge of the runtime layout of arrays in memory, whereas they have much less information about complex data structures allocated on the heap. In such (pointer-based) data structures, both the relative layout of *distinct data structures* in memory (which affects working set sizes) and the relative layout of nodes *within a single data structure* (which affects memory traversal patterns) are difficult to predict. One direct consequence is that irregular memory traversal patterns often have worse performance, both because of poor spatial locality and because techniques like hardware stride prefetching are not effective. A potentially more far-reaching consequence of the lack of layout information is that many compiler techniques (e.g., software prefetching, data layout transformations, and safety analysis) are either less effective or not applicable to complex data structures.

Despite the potential importance of data structure layouts, compiler transformations for pointer-intensive programs are performed primarily using pointer and dependence analysis, and *not by controlling and using information about the layout of pointer-based data structures*.

Several compiler techniques attempt to modify the layout of pointer-based data structures by giving hints or memory layout directives to a runtime library [12], memory allocator [9], or garbage collector [28, 29]. None of these techniques attempt *to extract information about or to control* the relative layouts of objects within a data structure or of distinct data structures, nor can they be directly extended to do so. Reliable data layout information and control are necessary for data layout properties to be used as a basis for further compiler transformations.

An alternative approach for segregating heap objects under compiler control is the work on automatic region inference for ML [45, 44, 24] and more recently for Java [14, 10]. These techniques partition objects into heap regions based on lifetimes, with the primary goal of providing automatic memory management with little or no garbage collection for type-safe languages. In contrast, our primary goal in this work is to improve program performance by segregating heap data structures and by enabling further layout-based optimizations on these data structures (in fact, we do not try to reclaim memory automatically, except in limited cases). Because of their different goals, these techniques do not explore how to exploit data structure partitioning to optimize memory hierarchy performance and do not support non-type-safe languages like C and C++, which are important for many performance-sensitive applications. These previous approaches are compared with our work in more detail in Section 10.

This paper describes *Automatic Pool Allocation*, a transformation framework for arbitrary imperative programs that *segregates*

PLDI'05, June 12–15, 2005, Chicago, Illinois, USA.

distinct instances of pointer-based data structures in the heap into seperate memory pools, and allows different heuristics to be used to partially control the internal layout of those data structures. For example, each distinct instance of a list, tree, or graph identified by the compiler would be allocated to a separate pool. The paper also describes several simple optimizations that exploit the partitioning of data structures on the heap to further improve program performance. These optimizations are possible because Automatic Pool Allocation is a rigorous transformation performed by the compiler. In other work, we have used Automatic Pool Allocation and its underlying pointer analysis to develop other new, compiler techniques operating at the "macroscopic" level, i.e., at the level of entire data structures (rather than individual pointers or objects). These include techniques for pointer compression [35] and memory safety without garbage collection [19]. These techniques are very briefly summarized in Section 7. The goal of this paper is to describe and evaluate the pool allocation transformation itself and simple optimizations directly based on it.

More specifically, Automatic Pool Allocation takes a context-sensitive, field-sensitive points-to graph representation and partitions the heap so that objects represented by the same points-to graph node are allocated in a common pool[1]. By using a context-sensitive analysis with "heap cloning," distinct data structure instances that are created and processed by the same functions can be segregated into different pools. The lifetime of each pool is determined by bounding the lifetime of pointers to objects in that pool. The end result of pool allocation is a partitioning of the runtime heap into pools, a transformed program that allocates and frees memory from pools, and a mapping from each static pointer to the pool it points into. Based on this information, subsequent optimizations and analyses may be applied to the program.

The Automatic Pool Allocation algorithm supports arbitrary C and C++ programs, including programs with function pointers and/or virtual functions, recursion, varargs functions, non-type-safe memory accesses (e.g., via pointer casts and unions), setjmp/-longjmp, and exceptions. One of the key strengths of the algorithm is a simple strategy for correctly handling indirect calls, which is difficult because different functions called via a function pointer may have different allocation and deallocation behavior and because (in C or C++) may even have different signatures. The algorithm solves these complex issues via a relatively simple graph transformation phase, while keeping the code transformation process essentially unchanged. The transformation works correctly for incomplete programs, by only pool allocating memory that does not escape the scope of analysis.

Automatic Pool Allocation can directly improve program performance in several ways. First, since programs typically traverse and process only one or a few data structures at a time, segregating logical data structures reduces the memory working sets of programs, potentially improving both cache and TLB performance. Second, in certain cases, the allocation order within each data structure pool will match the subsequent traversal order (e.g., if a tree is created and then processed in preorder), improving spatial locality. Intuitively, both benefits arise because the layout of individual data structures is unaffected by intervening allocations for other data structures, and less likely to be scattered around in the heap. Third, in some cases, the traversal order may even become a simple linear stride, allowing more effective hardware prefetching than before. Note that Automatic Pool Allocation can also potentially hurt performance in two ways: by separating data that are frequently accessed together and by allocating nearly-empty pages to small pools (some of the techniques described later are intended to address these issues).

This paper describes several optimizations based on pool allocation that further improve program performance. First, we show that in certain cases, individual `free` operations on objects in a pool can be eliminated and the entire memory for the pool reclaimed when the pool is destroyed (without increasing memory consumption). Second, we describe several customized memory management choices that can be used at run time for pools with specific characteristics. The key to some of these optimizations is that different logical data structures tend to be used in different but well-defined ways that can be exploited, whereas simply segregating by type, lifetime, or runtime profile information would not typically be sufficient to apply all these optimizations.

We evaluate the performance impact of Automatic Pool Allocation and the subsequent optimizations, using heap-intensive benchmarks from the SPEC, PtrDist [2], Olden [39] and FreeBench [40] benchmark suites, and a few standalone applications. We find that many of these programs speed up by 10-25%, two by about 2x and two small benchmarks by more than 10x. Other programs are unaffected, and importantly, none are hurt significantly by the transformation. We also show that the total compile time for the transformation (including the context-sensitive pointer analysis that computes its input points-to graphs) is very small, requiring 1.25 seconds or less for programs up to 100K lines of source code. Finally, the subsequent optimizations contribute improvements (over pool allocation alone) by 10-40% for eight cases and 0-10% for the others. We also show that cache and/or TLB performance improves significantly in all case with significant speedups, and in many cases hit rates are improved roughly similarly at all levels of the memory hierarchy (L1 and L2 caches and TLB), indicating that the performance improvements are primarily due to reduced working sets.

Overall, this paper makes the following contributions:

- We propose a novel approach to improving performance of pointer intensive programs: segregating and controlling heap layout of pointer-based data structure instances and using further compiler optimizations that exploit this layout information.
- We present a new compiler algorithm for partitioning heap objects in C and C++ programs that is based on a context-sensitive pointer analysis, including a novel and simple strategy for correct handling of indirect function calls in arbitrary (including non-type-safe) programs.
- We present several simple but novel optimizations that optimize the performance of individual data structure pools based on their specific patterns of memory deallocation or type information for pool contents. In previous work [19, 35], we have demonstrated other uses of Automatic Pool Allocation as well.
- We present a detailed experimental evaluation of the performance impact of Automatic Pool Allocation, showing that the transformation can significantly improve memory hierarchy performance and overall running times of heap-intensive programs, and that the transformation has very low compilation time in practice.

Section 2 defines the assumptions we make about the points-to graph input to the transformation. Section 3 describes the main pool allocation transformation, and Section 4 describes several important refinements. Sections 5 and 6 define a suite of simple pool optimizations and describe heuristics for pool collocation (respectively). Section 7 describes the key properties provided by Automatic Pool Allocation and two example clients, Section 8 describes our implementation, and Section 9 contains our experimental evaluation of the transformation. Finally, Section 10 contrasts this work with prior work in the field and Section 11 concludes the paper.

[1] Less aggressive pointer analyses can also be used but may not distinguish data structure instances or may give less precise information about their internal structure.

```c
struct list { list *Next; int *Data; };
list* createnode(int *Data) {
  list *New = malloc(sizeof(list));
  New->Data = Data;
  return New;
}
void splitclone(list *L, list **R1, list **R2) {

  if (L == 0) { *R1 = *R2 = 0; return; }
  if (some_predicate(L->Data)) {
    *R1 = createnode(L->Data);
    splitclone(L->Next, &(*R1)->Next, R2);
  } else {
    *R2 = createnode(L->Data);
    splitclone(L->Next, R1, &(*R2)->Next);
  }}
int processlist(list* L) {
  list *A, *B, *tmp;

  // Clone L, splitting nodes in list A, and B.
  splitclone(L, &A, &B);
  processPortion(A);  // Process first list
  processPortion(B);  // process second list

  // free A list
  while (A) { tmp = A->Next; free(A); A = tmp; }
  // free B list
  while (B) { tmp = B->Next; free(B); B = tmp; }

}
```

(a) Input C program manipulating linked lists

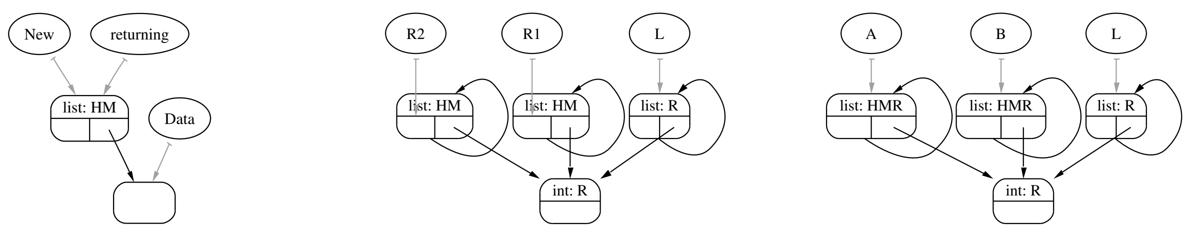
```c
struct list { list *Next; int *Data; };
list* createnode(Pool *PD, int *Data) {
  list *New = poolalloc(PD, sizeof(list));
  New->Data = Data;
  return New;
}
void splitclone(Pool *PD1, Pool *PD2,
                list *L, list **R1, list **R2) {
  if (L == 0) { *R1 = *R2 = 0; return; }
  if (some_predicate(L->Data)) {
    *R1 = createnode(PD1, L->Data);
    splitclone(PD1, PD2, L->Next, &(*R1)->Next, R2);
  } else {
    *R2 = createnode(PD2, L->Data);
    splitclone(PD1, PD2, L->Next, R1, &(*R2)->Next);
  }}
int processlist(list* L) {
  list *A, *B, *tmp;  Pool PD1, PD2;
  poolcreate(&PD1, sizeof(list), 8);
  poolcreate(&PD2, sizeof(list), 8);
  splitclone(&PD1, &PD2, L, &A, &B);
  processPortion(A);  // Process first list
  processPortion(B);  // process second list

  // free A list: this loop is eventually eliminated
  while (A) { tmp = A->Next; poolfree(&PD1, A); A = tmp; }
  // free B list this loop is eventually eliminated
  while (B) { tmp = B->Next; poolfree(&PD2, B); B = tmp; }
  pooldestroy(&PD1); pooldestroy(&PD2);  // destroy pools
}
```

(b) C code after the basic pool allocation transformation

Figure 1. Example illustrating the Pool Allocation Transformation

'processlist' copies a list into two disjoint lists (based on some predicate), processes each, then frees them. After basic pool allocation, the new lists are put in separate pools ($PD1$ and $PD2$) which are each contiguous in memory. After subsequent optimization, the calls to `poolfree` and the loops containing them are removed because `pooldestroy` frees all pool memory.



(a) DS Graph for createnode (b) DS Graph for splitclone (c) DS Graph for processlist

Figure 2. BU DSGraphs for functions in Figure 1 (a)

2. Background: Points-To Graph & Example

In this section, we briefly introduce the running example used by this paper and specify the points-to graph representation and properties that are used by our description of Automatic Pool Allocation and its optimizations. Figure 1(a) shows the running example we use to illustrate the pool allocation transformation. This program fragment traverses an input list (L), creating two new lists (A and B) based on the input list, processes the new lists separately, and finally deallocates them.

Automatic Pool Allocation is driven by a points-to graph computed by some pointer analysis that uses an explicit representation of memory [27]. In our implementation, we use an algorithm we call Data Structure Analysis (DSA) [32] to compute these points-to graphs. DSA is *context-sensitive* (both in its analysis and in that it distinguishes heap and stack objects by entire acyclic call paths), *unification-based*, and *field-sensitive*, and we believe these properties are important for pool allocation for the reasons explained below. We have shown that DSA is both extremely fast and scalable (it can analyze programs of 220K lines of code like 176.gcc in under 2 seconds [32]), and requires a small fraction of the compilation time taken by "`gcc -O3`."

Context-sensitive naming of heap objects by (acyclic) call paths is required to distinguish data structure instances that may be created, processed, or destroyed by calling common functions. For example, this property enables DSA to determine that lists A and B are disjoint in Figure 1(a) even though their nodes are allocated at a common allocation site, enabling pool allocation to assign them to distinct pools. With less or no context-sensitivity (e.g., if heap objects were distinguished only by allocation site), such data structures would not be segregated into distinct pools.

A **unification-based** pointer analysis [43] merges the potential targets of a pointer into a single set of objects , yielding a points-to graph where every pointer variable or pointer field points to at most one node. We use a unification-based approach for two reasons. First, as we argued in [32], it is crucial to making DSA both extremely fast and scalable despite distinguishing memory objects by call paths. Second, it greatly simplifies Automatic Pool Allocation because it ensures that every pointer points to a unique node and hence a unique pool. Pool allocation without unification would require a mechanism (e.g., "fat pointers") to track the pool pointed to by each pointer value at run time, which is significantly more complex and can hurt both performance and compatibility with external libraries. Furthermore, there is some evidence that adding context-sensitivity substantially reduces the precision advantage of subset-based over unification-based algorithms [36, 16, 32].

A **field-sensitive** pointer analysis distinguishes the targets of distinct pointer fields within a record. This is important in practice for pool allocation with a unification-based algorithm because merging the targets of unrelated pointer fields in a node would lose most of the internal structural information for multi-level pointer-based data structures. DSA is field-sensitive for memory objects that are accessed in a type-consistent way, a property which it infers during the analysis.

While we used DSA in our work, we know several other context-sensitive analyses [20, 37, 10, 38] which provide all of the properties required by the transformation or could be extended to do so. We describe the properties of the points-to graphs and other information required by the Automatic Pool Allocation transformation below.

2.1 Points-to Graph Assumptions

This section describes the properties of the points-to graphs used by Automatic Pool Allocation, which we term Data Structure (or DS) Graphs. A DS graph is a compile-time description of the memory objects created by a function or program and their points-to properties. We assume that a DS graph is computed for each function, representing the memory objects reachable from variables in that function or from global variables. For reference, Figure 2 show the graphs computed by DSA for the functions in the example.

Formally, a **DS Graph** is a directed multi-graph, where the nodes and edges are defined as follows:

DS Node: A DS node is a 5-tuple $\{\tau, F, M, A, G\}$. τ is some (program-defined) scalar, array, record or function type, or $\perp$ representing an unknown type. In the transformation, $\perp$ is treated like an unknown-size array of bytes (the A flag, described below, is set to true when $\tau = \perp$). F is an array of fields, one for each possible field of the type τ. Scalar types and $\perp$ have a single field, record types have a field for each element of the record, array types are treated as their element type (i.e. array indexing is ignored), and functions do not have fields. M is a set of memory classes, written as a subset of $\{\mathbf{H}, \mathbf{S}, \mathbf{G}, \mathbf{U}\}$, indicating Heap, Stack, Global and Unknown memory objects respectively (multiple flags can be set on one node due to the use of unification). A node with $\mathbf{U} \in M$ is assigned type $\perp$. Finally, if $\mathbf{G} \in M$, then G is a non-empty set of global variables and functions included in the objects for this node; otherwise, G is empty. A is a boolean that is true if the node includes an array object. Finally, though this paper does not use the information, DSA also infers Mod and Ref information, which are shown as "**M**" and "**R**" in the figures.

DS Edge: A DS edge is a 4-tuple $\{s, f_s, t, f_t\}$. s and t are DS nodes, while f_s and f_t are fields of s and t respectively. Thus, the graph provides a field-sensitive representation of points-to information. A field of a node may lack an outgoing DS edge only if the field is known not to contain a pointer type, e.g., if the node represents a function (the function itself doesn't point to anything else), is a floating point or small integer type, or if $M = \{\mathbf{U}\}$. In this paper, we use the notation "N(ptr)" to indicate the node which the scalar pointer ptr points to.

Figure 2(b) shows the DS graph computed by our compiler for function `splitclone` of the example. Note that each node of type `list` has two fields[2]. The cycles indicate recursive data structures.

[2] The diagrams in this paper show pointers to nodes in cases where the pointer targets the first field of the node, due to limitations of the graph layout tool we use.

```
void poolcreate(Pool* PD, uint Size, uint Align)
      Initialize a pool descriptor.
void pooldestroy(Pool* PD)
      Release pool memory and destroy pool descriptor.
void* poolalloc(Pool* PD, uint numBytes)
      Allocate an object of numBytes bytes.
void poolfree (Pool* PD, void* ptr)
      Mark the object pointed to by ptr as free.
void* poolrealloc(Pool* PD, void* ptr, uint numBytes)
      Resize an object to numBytes bytes.

void poolinit_bp(Pool *PD, uint Align)
      Initialize a bump-pointer pool descriptor.
void *poolalloc_bp(Pool *PD, uint NumBytes)
      Allocate memory from a bump-pointer pool.
void pooldestroy_bp(Pool *PD)
      Release a bump-pointer pool.
```

Figure 3. Interface to the Pool Allocator Runtime Library

$R1$ and $R2$ point to distinct nodes, indicating that the two linked lists are completely disjoint.

There are two other assumptions about DS graphs used in this work. First, we assume that the DS Graph for each function includes information about that function and all of the functions that it calls (but no information about its callers). Section 3.3 explains why this assumption is safe. For example, a node with $\mathbf{H} \in M$ indicates heap objects allocated or freed in the current function or its callees, but not its callers. Several context-sensitive analyses, including DSA and others [37, 38], compute separate "Bottom-Up" (BU) and "Top-Town" (TD) graphs: the Bottom-Up graphs capture exactly the information required. For example, the graph in Figure 2(b) incorporates the points-to, mod/ref, and flag effects of both calls to "createnode": it includes two copies (one for each call) of the H flag and the edge from the list node to the integer data node present in Figure 2(a).

Second, there are a few primitive operations on DS graphs used in the transformation, including merging two graphs and matching nodes between graphs for a callee and a caller. These are defined and explained where they are used in the next Section.

3. The Core Transformation

The pool allocation transformation operates on a program containing calls to `malloc` and `free`, and transforms the program to use a pool library, described below. The algorithm uses a points-to graph and call graph, both of which are computed by DSA in our implementation. The transformation is a framework which has several optional refinements. In this section, we present a "basic" version of the transformation in which all heap objects are allocated in pools (i.e., none are allocated directly via `malloc`) and every node in the points-to graph generates a separate static pool (explained below). In the next section, we discuss refinements to this basic approach.

3.1 Pool Allocator Runtime Library

Figure 3 shows the interface to the runtime library. Pools are identified by a pool descriptor of type `Pool`. The functions `poolalloc`, `poolfree`, and `poolrealloc` allocate, deallocate, and resize memory in a pool. The `poolcreate` function initializes a pool descriptor for an empty pool, with an optional size hint (providing a fast path for a commonly allocated size) and an alignment required for the pool (this defaults to 8, as in many standard malloc libraries). `pooldestroy` releases all pool memory to the system heap. The last three functions (with suffix "`_bp`") are variants that use a fast "bump pointer" allocation method, described in Section 5.

The library internally obtains memory from the system heap in blocks of one or more pages at a time using `malloc` (doubling the size each time). We implemented multiple allocation algorithms but the version used here is a general free-list-based allocator with coalescing of adjacent free objects. It maintains a four-byte header per object to record object size. The default alignment of objects (e.g., 4- or 8-byte) can be chosen on a per-pool basis, for reasons described in Section 5. The pool library is general in the sense that it does not require all allocations from a pool to be the same size.

3.2 Overview Using an Example

The basic pool allocation transformation is illustrated for the example program in Figure 1(b), which shows the results of our basic transformation in C syntax. The incoming list L and the two new lists have each been allocated to distinct pools (the pool for L is not passed in and so not shown; the new lists use pools PD1 and PD2). The list nodes for A and B will be segregated in the heap, unlike the original program where they will be laid out in some unpredictable fashion (and possibly interleaved) in memory. The items in each pool are explicitly deallocated and the pools are destroyed within `processList` when the data they contain is no longer live.

We can use this example to explain the basic steps of the transformation. The DS graphs are shown in Figure 2. First, we use each function's DS graph to determine which **H** nodes are accessible outside their respective functions, i.e., "escape" to the caller. The **H** nodes in `createnode` and `splitclone` do escape, because they are reachable from a returned pointer and a formal argument, respectively. The two in `processlist` (`A` and `B`) do not. The latter are candidates for new pools in `processlist`.

The transformation phase inserts code to create and destroy the pool descriptors for `A (PD1)` and `B (PD2)` in `processlist` (see Figure 1(b)). It adds pool descriptor arguments for every **H** node that escapes its function, i.e., for nodes pointed to by `R1` and `R2` in `splitclone` and the node pointed to by `New` in `createNode`. It rewrites the calls to `malloc` and `free` with calls to `poolalloc` and `poolfree`, passing appropriate pool descriptors as arguments. Finally, it rewrites other calls to (e.g., the calls to `splitclone` and `createnode`) to pass any necessary pool descriptor pointers as arguments. At this point, the basic transformation is complete.

Further refinements of the transformation move the `pooldestroy` for `PD1` as early as possible within the function `processlist`, and then eliminate the calls to free items in the two lists (since these items will be released by pooldestroy before any new allocations from any pool) and hence the loop enclosing those calls to free. The final resulting code (Figure 8) puts each linked list into a separate pool on the heap, made the list objects of each list contiguous in memory, and reclaims all the memory for each list at once instead of freeing items individually. In the example, the list nodes are placed in dynamic allocation order within their pool.

3.3 Analysis: Finding Pool Descriptors for each H Node

The analysis phase identifies which pool descriptors must be available in each function, determines where they must be created and destroyed, and assigns pool descriptors to DS nodes. We use the term *static pool* to refer to a single `poolcreate` statement in the generated code. By definition, $\mathbf{H} \in M$ for a node if the objects of that node are returned by `malloc` or passed into `free` by the current function or any of its callees, since we assume a Bottom-up DS graph (see Section 2.1). These identify exactly those nodes for which a pool descriptor must be *available* in the current function.

Automatic Pool Allocation computes a map (pdmap) identifying the pool descriptor corresponding to each DS node with $\mathbf{H} \in M$. We initially restrict pdmap to be a one-to-one mapping from DS nodes to pool descriptor variables; Section 6 extends pdmap to allow a many-to-one mapping. We must handle two cases: 1) the pool escapes the current function and 2) the pool lifetime is bound by the function. In the first case, we add a pool descriptor argument to the function, in the second, we create a descriptor on the stack for the function and call `poolcreate`/`pooldestroy`. These two cases are differentiated by the "escapes" property for the DS node.

The "escapes" property is determined by a simple escape analysis on the bottom-up DS graphs, implemented as a depth-first traversal. In particular, a node escapes iff 1) a pointer to the node is returned by the function (e.g. `createnode`) 2) the node is pointed to by a formal argument (e.g. the R1 node in `splitclone`) 3) the node is pointed to by global variable and the current function is not main, or 4) (inductively) an escaping node points to the node.

A subtle point is that any node that does not escape a function will be unaffected by callers of the function, since the objects at such a node are not reachable (in fact, may not exist) before the current function is called or after it returns. *This explains why it is safe to use a BU graph for pool allocation*: Even though the BU graph does not reflect any aliases induced by callers, the non-escaping nodes are correctly identifiable and all information about them is complete, including their type τ, incoming points-to edges, and flags. In fact, in DSA, the escapes property is explicitly computed and all non-escaping nodes are marked using a "**C**"omplete flag [32]. It can be computed easily using the above definition by any context-sensitive algorithm that has similar points-to graphs.

3.3.1 The Basic Transformation

Figure 4 shows the pseudocode for a basic version of the Automatic Pool Allocation transformation, which does not handle indirect function calls. The algorithm makes two passes over the functions in the program in arbitrary order. The first (lines 1–11) adds arguments to functions, creates local pool descriptors, and builds the pdmap. The second (lines 12–20) rewrites the bodies of functions using pdmap.

```
   basicpoolallocate(program P)
1    ∀F ∈ functions(P)
2      dsgraph G =DSGraphForFunction(F)
3      ∀n ∈ nodes(G)                  // Find pooldesc for heap nodes
4        if (H ∈ n.M)
5          if (escapes(n))             // If node escapes fn
6            Pool* a = AddPoolDescArgument(F, n)
7            pdmap(n) = a              // Remember pooldesc
8            argnodes(F) = argnodes(F) ∪ {n}
9          else                        // Node is local to fn
10           Pool* pd = AddInitAndDestroyLocalPool(F, n)
11           pdmap(n) = pd
12   ∀F ∈ functions(P)
13     ∀I ∈ instructions(F)     // Rewrite function
14       if (I isa 'ptr = malloc(size)')
15         replace I with 'poolalloc(pdmap(N(ptr)), size)'
16       else if (I isa 'free(ptr)')
17         replace I with 'poolfree(pdmap(N(ptr)), ptr)'
18       else if (I isa 'call Callee(args)')
19         ∀n ∈ argnodes(Callee)
20           addCallArgument(pdmap(NodeInCaller(F, I, n)))
```

Figure 4. Pseudo code for basic algorithm

For each node that needs a pool in the function, the algorithm either adds a pool descriptor argument (if the DS node escapes) or it allocates a pool descriptor on the stack. Non-escaping pools are initialized (using `poolcreate`) on entry to the function and destroyed (`pooldestroy`) at every exit of the function (these placement choices are improved in Section 4.2). Because the DS node does not escape the function, we are guaranteed that any memory allocated from that pool can never be accessed outside of the current function, i.e., it is safe to destroy the pool, even if some memory was not deallocated by the original program. Note that this may actually eliminate some memory leaks in the program!

In the second pass (lines 12–20), the algorithm replaces calls to `malloc()` and `free()`[3] with calls to `poolalloc` and `poolfree`.

[3] Note that "malloc wrappers" (like `calloc`, `operator new`, `strdup`, etc) do not need special support from the pool allocator. Their bodies are simply linked into

We pass the appropriate pool descriptor pointer using the pdmap information saved by the first pass. Since the DS node must have an **H** flag, a pool descriptor is guaranteed to be available in the map.

Calls to functions other than `malloc` or `free` must pass additional pool descriptor arguments for memory that escapes from them. Because the BU Graph of the callee reflects all accessed memory objects of all transitive callees, any heap objects allocated by a callee will be represented by an **H** node in the caller graph (this is true even for recursive functions like `splitclone`). This property guarantees that a caller will have all of the pool descriptors that any callee will ever need.

A key primitive computable from DS graphs is a mapping, $NodeInCaller(F, C, n)$. For a call instruction, C, in a function F, if n is a DS node in any possible callee at that call site, then $n' = NodeInCaller(F, C, n)$ identifies the node in the DS graph of F corresponding to node n due to side-effects of the call C (i.e., n' includes the memory objects of node n visible in F due to this call). The mapping is computed in a single linear-time traversal over matching paths in the caller and callee graphs, starting from matching pairs of actual and formal nodes, matching pairs of global variable nodes, and the return value nodes in the two graphs if any. If n escapes from the callee, then the matching node n' is guaranteed to exist in the caller's BU graph (due to the bottom-up inlining process used to construct the BU graphs), and is unique because the DS graphs are unification-based [32].

Identifying which pool of the caller (F) to pass for callee pool arguments at call instruction I is now straightforward: for each callee node n that needs an argument pool descriptor, we pass the pool descriptor for the node $NodeInCaller(F, I, n)$ in the caller's DS graph. We record the set of nodes ("argnodes") that must be passed into each function, in the first pass.

Variable-argument functions do not need any special treatment in the transformation because of their representation in the BU graphs computed by DSA. In particular, the DS graph nodes for all pointer-compatible arguments passed via the "`...`" mechanism (i.e., received via `va_arg`) are merged so that they are represented by a single DS node in the caller and callee. If the DS node pointed to by this argument node has $\mathbf{H} \in M$, a single pool argument is added to the function. At every call site of this function, the nodes for the actual argument (corresponding to the merged formals) will also have been merged, and the pool corresponding to this node will be found by $NodeInCaller(F, I, n)$ and passed in as the pool argument. Note that explicit arguments before the `...` are not merged and can have distinct pools.

3.3.2 Passing Descriptors for Indirect Function Calls

Indirect function calls make it much more complex to pass correct pool descriptor arguments to each function. There are multiple difficulties. First, different functions called via a function pointer at the same call site may require different sets of pools. Figure 5 shows a simple example where `func1` needs no pools but `func2` needs one pool, and both are called at the same site. Second, different indirect call sites can have different but overlapping sets of callees, e.g., $\{F_1, F_2\}$ and $\{F_2, F_3\}$ at two different call sites. In order to avoid cloning F_2 into two versions, we must pass the same pool arguments to all three functions F_1, F_2 and F_3. This raises a third major problem: because the call graph says that F_3 is not a callee at the first call-site, its DS graph was never inlined into that of the caller at that call-site. This means that the matching of nodes between caller and callee graphs, which is essential for passing pool descriptors, may be undefined: $NodeInCaller(F, C, n)$ may not exist for all escaping n. Programs that violate the type signatures

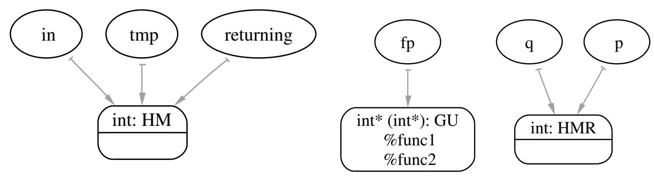
```
int* func1(int* in) { *in = 1; return in; }
int* func2(int* in) { free(in);
                      in = (int*) malloc(sizeof(int));
                      *in = 2; return in; }
int caller(int X) {
  int* (*fp)(int*) = (X > 1)? func1 : func2;
  int *p = (int*) malloc(sizeof(int));
  int *q = fp(p);
  return *q;
}
```

(a) Input C program with an indirect function call

```
int* func1(Pool* P, int* in) { *in = 1; return in; }
int* func2(Pool* P, int* in) { poolfree(P, in);
              in = (int*) poolalloc(P, sizeof(int));
              *in = 2; return in; }

int caller(int X) {
  Pool PD1;  poolcreate(&PD1, ...);
  int* (*fp)(int*) = (X > 1)? func1 : func2;
  int *p = (int*) poolalloc(PD1, sizeof(int));
  int *q = fp(PD1, p);
  pooldestroy(&PD1); return *q;
}
```

(b) C code after pool allocation



(c) Merged EBU Graph for `func1` and `func2`

(d): EBU Graph for `caller`

Figure 5. Example with function pointers

Though `func1` and `func2` are called at the same call site, only one needs a pool descriptor. The algorithm puts them in a single equivalence class, merges their DS graphs, adds a pool argument to both functions.

of functions at call sites (not uncommon in C code) exacerbate all three problems because any attempt to match pool arguments explicitly for different callees must account for mismatches between the actual and formals for each possible callee.

Our solution is composed of two key principles, described below, and shown in pseudocode in Figure 6. The first principle is to partition into equivalence classes so that all potentially callees at an indirect call site are in the same class. We then treat *all* functions in the same equivalence class as potential callees for that call site. For example, `func1` and `func2` in the example figure are put into the same class, and so are F_1, F_2 and F_3 in the example above. Lines 1-2 uses the call graph to partition all the functions of the program into disjoint equivalence classes in this manner.

The second principle is to simplify matching nodes between different callees at a call site with the nodes of the caller by merging the graphs of all functions in an equivalence class, and then updating the caller graphs to be consistent with the merged callee graphs. Merging the graphs ensures that an identical set of pool descriptor formal arguments will be inferred for all functions in the class. Updating the caller graphs to be consistent with the callee graphs (as explained below) ensures that the third problem above — finding matching nodes between callee and caller — is always possible.

In the example, the algorithm merges the DS graphs of `func1` and `func2` into the common graph shown in Figure 5(c), and uses this common graph to transform both functions. This results in a matching set of pool arguments for both functions, even though the

the program and treated as if they were a user function, getting new pool descriptor arguments to indicate which pool to allocate from.

pool will be unused in `func1`. This common graph is merged into the caller, resulting in the graph shown in Figure 5(d). Using this graph, one descriptor is passed to both functions at the call site.

The implementation of these graph merging and inlining steps (lines 3-8 of Figure 6) use two primitive DSA operations – merging two graphs and performing a bottom-up inlining pass on strongly-connected components (SCCs) of the call graph. To merge the graphs of two functions in an equivalence class (lines 3-4), we copy one graph into the other, then unify corresponding formal argument nodes (ignoring any extra nodes in one of the graphs if the formal argument lists do not match), global nodes, and the return value node of each graph. Unifying nodes causes recursive merging and can potentially cause loss of some type information if merged nodes have incompatible types.

Finally, we perform a bottom-up "inlining" pass on the strongly connected components (SCCs) of the call graph, inlining merged graphs of the callees into their callers. This simply requires repeating the bottom-up inlining pass of the DSA algorithm (starting with the merged equivalence-class graphs of each function). This step is described in detail in [32].

We call the resulting DS graphs the EBU ("equivalence bottom-up") graphs. The EBU graph is more conservative than the original DS graph because functions known not to be called at a call-site may be merged into the caller along with those that are (because they are in the same equivalence class). Such cases do not arise often in practice, and the merging of equivalence class graphs greatly simplifies the overall transformation algorithm by solving the above three problems with a uniform strategy based on existing DS graph primitives.

```
  completepoolallocate(program P)
1   ∀cs ∈ callsites(P)                          // Build equivalence classes
2     unify equivclasses(callees(cs))
3   ∀ec ∈ equivclasses(functions(P))           // Build graph for each class
4     ECGraph(ec) = mergeGraphs(DSGraphs(members(ec)))
5   ∀scc ∈ tarjansccfinder(callgraph(P))
6     ECGraph(scc) = mergeGraphs(ECGraphs(functions(scc)))
7     ∀cs ∈ callsites(scc)                     // Inline callees into caller
8       ECGraph(scc) = mergeGraph(cs, ECGraph(callees(cs)))
9   basicpoolallocate(P)
```

Figure 6. Pseudo code for complete pool allocator

Given the EBU graphs for a program, the pool allocator is now guaranteed to have all of the pool descriptors required at an indirect call site for any of the potential callees of the call site, allowing it to apply the `basicpoolallocate` algorithm safely. Note that lines 17-19 simply have to use the common graph for all callees even though there may now be multiple callers for the call at line 17.

A detailed discussion of the complexity of the Automatic Pool Allocation algorithm is outside the scope of this work but is available in [32]. Briefly, all parts of the algorithm with the exception of lines 5-8 of Figure 6 are linear in the total size of all DS graphs and the number of instructions in the program, and $\Theta(n\alpha(n))$ in the number of call graph edges. The complexity of lines 5–8, the EBU phase, is similar to the BU phase of DSA, i.e., $\Theta(n\alpha(n)+k\alpha(k)c)$, if n, k and c denote the total number of instructions, the maximum size of a DS graph for a single procedure, and the number of edges in the call graph. In practice, k is very small, typically on the order of a hundred nodes or less, even for large programs [32].

4. Algorithm Refinements and Implementation

4.1 Argument Passing for Global Pools

A DS node reachable from a global variable requires a pool created in `main` because the heap objects at that node may be live throughout the lifetime of the program. This introduces a major source of runtime overhead because such a pool would have to passed down through many layers of function calls to be available in each function that actually allocates or frees data in the pool. In practice, we have found that programs which have many heap nodes reachable from globals may get thousands of arguments added to the program.

The solution is simple: we create a global variable to hold the pool descriptor for each heap node reachable from a global and use this where needed, instead of passing the pool descriptor in via function arguments. In practice, this refinement greatly reduces the number of pool arguments that must be passed to functions in some C programs. Most importantly, it ensures that the only pool arguments that must be passed to a function are for nodes reachable from pointers passed in as function arguments, making the number of pool arguments grow with the number of formal pointer arguments in the original function.

```
int processlist(list* L) {
  list *A, *B, *tmp;
  Pool PD1, PD2; // initialize pools
  poolcreate(&PD1, ...); poolcreate(&PD2, ...);
  splitclone(&PD1, &PD2, L, &A, &B);
  processPortion(A); // Process first list
  processPortion(B); // process second list

  while(A) { tmp=A->Next; poolfree(&PD1, A); A=tmp; }
  pooldestroy(&PD1);   // NOTE: this moved up

  while(B) { tmp=B->Next; poolfree(&PD2, B); B=tmp; }
  pooldestroy(&PD2);   // destroy pool PD2
}
```

Figure 7. After moving `pooldestroy(&PD1)` earlier

```
int processlist(list* L) {
  list *A, *B, *tmp;
  Pool PD1, PD2;
  poolcreate(&PD1, ...); poolcreate(&PD2, ...);
  splitclone(&PD1, &PD2, L, &A, &B);
  processPortion(A); // Process first list
  processPortion(B); // process second list
  pooldestroy(&PD1); // destroy pool (including nodes)
  pooldestroy(&PD2); // destroy pool (including nodes)
}
```

Figure 8. After eliminating `poolfree` calls and dead loops

4.2 poolcreate/pooldestroy Placement

The algorithm described above places `poolcreate`/`pooldestroy` calls at the entry and exits of each function. In practice, the lifetime of the data objects in a pool may begin at a later point in the function and may end before the end of the function. Moving the pool create/destroy calls later and earlier within the function reduces the lifetime of objects in the pool. This refinement can also make it more likely that the refinement in Section 4.3 can apply.

We modified the basic algorithm so that it initially does not insert `poolcreate / pooldestroy` calls but performs all other transformations. For each pool that must be created in a function, we use two simple depth-first traversals of the CFG to identify all basic blocks where the pool descriptor must be live, based on its uses, and then place `poolcreate`/`pooldestroy` calls at edges entering or leaving a live block from or to a non-live block. The overall algorithm is extremely simple and linear in the size of CFG.

Figure 7 illustrates this placement for the `processlist` function in our example. The call to `pooldestroy(&PD1)` has been moved earlier in the function, to immediately after the `while` loop that reads the Next field from nodes in PD1 pool. The `poolcreate` calls for both pools cannot be moved any later.

In general, the `poolcreate` and `pooldestroy` calls can be moved interprocedurally to further reduce the lifetime of pools, similar to Aiken et al.'s work [1]. However, that would likely require a more expensive, flow-sensitive interprocedural algorithm [1] and we have not attempted this so far.

4.3 `poolfree` Elimination

The final refinement is to eliminate unnecessary `poolfree` calls. Many short-lived data structures have a "build-use-destroy" pattern, in which all allocations happen before any deallocations. For example, consider Figure 7. Between the call to `poolfree(&PD1, A)` and the call to `pooldestroy(&PD1)`, there are no allocations out of *any* pool. This means that it is unnecessary to release the memory in pool PD1 any earlier than the `pooldestroy(&PD1)`, when all the memory of the pool will be released back to the system. We eliminate the call to `poolfree(&PD1, A)`, which also allows the compiler to eliminate the enclosing loop (similarly for `PD2`). Effectively, we have performed a simple kind of *static garbage collection* for the objects in this pool [31]. Note that moving `pooldestroy` calls earlier in the code can increase the opportunities for finding candidate `poolfree` calls to eliminate.

Again, we implemented this optimization as a simple, backward dataflow analysis on the CFG, without interprocedural information. The analysis looks for any occurrence of `poolfree(P)` such that *no* path from the `poolfree` to the `pooldestroy` calls for P contains any allocation out of any pool (including P). The result for `processlist` is shown in Figure 8.

5. Pool Allocation Optimizations

We describe four simple optimizations that exploit the partitioning of heap objects and the differences in behavior of different pools. The benefits of all four optimizations are evaluated in Section 9.

1) **Avoiding pool allocation for singleton objects:** Our simplest optimization avoids pool-allocating nodes that appear to be used for a single object. We identify such pools by finding **H** nodes not pointed to by any other memory object (including itself), e.g. they are only pointed to by local scalar variables. This optimization avoids creating and destroying a pool descriptor (minor) and avoids significant wasted space when the object is much smaller than the smallest internal page (potentially significant when many such pools are created). We term this "Selective PA".

2) **Eliminating `poolfree` operations:** The refinement described in Section 4.3 is an optimization that can eliminate the final `poolfree` operations of a data structure, which we term "PoolFreeElim." In some cases, this optimization can make entire data structure traversals dead, as in the example above. As noted, this optimization is enhanced by smart `pooldestroy` positioning.

Note that segregating data structures into distinct pools is what enables this optimization to exploit the "build-use-destroy" pattern shown by (some) data structures. For example, if there were any allocations for the second list between `pooldestroy(&PD1)` and the second `while` loop, the optimization would not be possible without separating the lists into distinct pools.

3) **"Bump-pointer" allocation:** If memory is never free'd back to a pool, there is no need for the pool library to maintain freelists or the 4-byte header on each object. The bump pointer optimization detects pools whose memory is only released by a pooldestroy, as explained below. It then changes the pool operations to use the "`_bp`" versions of the pool routines for such pools. This allocator has a shorter allocation path than the normal pool allocator and packs objects more densely in memory and cache (due to the missing object header). This optimization is clearly enhanced by the poolfree elimination optimization, which allows both pools in the example to be changed into bump pointer pools.

We implemented this as a simple post-pass over the program. Our implementation walks the use chain of each pool descriptor looking for any use that is not a `poolcreate`, `pooldestroy`, or `poolalloc`. If only these uses occur, the pool is promoted to use a bump pointer by replacing the ordinary pool library calls with the "`_bp`" versions. Note that our implementation currently cannot promote any pools whose descriptors are passed into any other function, including user functions like those in the example.

4) **Intelligent object alignment:** A traditional `malloc` library must be conservative about memory alignment because it lacks type information for the memory it allocates. Many architectures (e.g., Sparc V9) *require* that certain 8-byte values (e.g., C doubles) must be 8-byte aligned, while others (e.g., x86) impose significant performance penalties if such values are misaligned. This forces many system malloc libraries to use 8-byte alignment for *all* objects, increasing memory consumption and reducing cache density. For example, two successive 16 byte objects will be placed 24 bytes apart because 4 bytes are typically used as a malloc object header, forcing an extra 4 bytes of padding per object for proper alignment.

Because many pools are type homogeneous, we have reliable compile-time information about data types in such pools. Therefore, we use 4-byte alignment when it is provably safe (i.e., a pool is type homogenous *and* no field of the object will be improperly aligned for the target architecture). Otherwise we use 8-byte alignment. The alignment is specified when the pool is created.

6. Node Collocation Heuristics

The pool allocation algorithm so far provides a framework for segregating heap data structures but never collocates objects of two DS nodes into the same pool. We can adapt the algorithm to collocate a set of nodes by changing line 9 so that `poolcreate` and `pooldestroy` are inserted for only one of the nodes, and initializing pdmap to use the same descriptor for the other nodes.

Since heap objects are laid out separately and dynamically within each pool, collocating objects can give the compiler some control over internal layout of data structures. Even more sophisticated control might be possible using additional techniques (e.g., [9]) customized on each pool.

We propose two example static options for choosing which **H** nodes should share a common pool. We define a *Collection* to be either a node with $A = true$, or any non-trivial strongly connected component (i.e., containing at least one cycle) in the DS Graph. Given this, any **H** node reachable from a Collection represents a set of objects that may be visited by a single traversal over the objects of the Collection.

OnePoolPerCollection: All candidate **H** nodes in a collection are assigned to a single pool. Any other **H** node reachable from a collection (without going through another collection) is assigned to the same pool as the collection. This choice effectively partitions the heap so that each minimal "traversable" collection of objects becomes a separate pool. Intuitively, this gives the finest-grain partitioning of recursive data structures, which are often hierarchical. It favors traversals over a single collection within such a hierarchical (i.e., multi-collection) data structure.

MaximalDS: A maximal connected subgraph of the DS graph in which all nodes are **H** nodes are assigned to a single pool. This partition could be useful as a default choice if there is no information about traversal orders within and across collections.

Our implementation supports flexible colocation policies, but in practice we have found that using a static collocation heuristics rarely outperform (and is often worse than) assigning each **H** node to a separate pool (see [32] for a discussion). We expect that more sophisticated static analysis of traversal patterns combined with profile data will be needed to statically choose the optimal colocation configuration. We leave it to future work to develop an effective approach for profitable collocation. The discussion above, shows, however that (a) collocation of pools can be implemented easily, and (b) qualitatively, the pointer analysis and pool allocation provide a useful basis for per-data-structure choices.

7. Compiler Applications of Pool Allocation

We believe that Automatic Pool Allocation combined with the underlying pointer analysis provides a new framework for analyzing and optimizing pointer-intensive programs, operating at the level of entire data structure instances, instead of individual load/store operations or individual data types. This is because Automatic Pool Allocation provides four fundamental benefits to subsequent compiler passes:

1. *Data structure-specific policies via segregation*: Allocating distinct data structures from different pools allows compiler and run-time techniques to be customized for each instance. These techniques can use both static pool properties (e.g., type information and points-to relationships) and dynamic properties (anything recordable in per-pool metadata).
2. *Mapping of pointers to pool descriptors*: The transformation provides a static many-to-one mapping of heap pointers to pool descriptors. This information is key to most transformations that exploit pool allocation because it enables the compiler to transform pointer operations into pool-specific code sequences. It is used by both the example applications described below.
3. *Type-homogeneous pools*: Many pools are completely type-homogeneous, as shown in Section 9, even C programs. Novel compiler and run-time techniques are possible for type-homogeneous pools that would not be possible on other pools or the general heap (e.g. the alignment optimization).
4. *Knowledge of the run-time points-to graph*: One way to view pool allocation is that it partitions the heap to provide a run-time representation of the points-to graph. The compiler has full information about which pools contain pointers to other pools and, for type-homogeneous pools, where all the intra-pool and inter-pool pointers are located. Such information is useful any time pointers need to be traversed or rewritten at run-time.

The optimizations described earlier show some simple examples of how compiler techniques can exploit these benefits. In other work, we developed two new compiler techniques that are much more sophisticated and exploit many or all of the above properties of pool allocation. We summarize these very briefly here; each is described in detail elsewhere [19, 35]:

Transparent Pointer Compression: After pool allocation, the static mapping of pointers to pool descriptors allows us to use *pool indices* (i.e., byte offsets relative to the pool base) instead of pointers to identify objects in a pool. Since most individual data structures are likely to have far less than 2^{32} distinct nodes, segregating data structure instances allows us to represent these indexes with integers smaller than the pointer size (e.g., 32 bits on a 64-bit host). In [35], we implement this transformation and show that it can have a significant and sometimes dramatic impact on both the performance and the memory footprint of pointer-intensive programs on 64-bit systems. This transformation exploits the segregation of data structures into pools (which allows small indices), type homogeneity (which allows compression of indices by rewriting structure accesses and allocations), and of course the mapping of pointers to pools (making the pool base implicit).

We also describe a dynamic version of the algorithm where the pool runtime library dynamically rewrites nodes to grows pointers in data structures when the $2^{32}nd$ node is allocated. This allows us to speculatively compress pointers to 32-bits while retaining the ability to dynamically expand them to 64-bits if full addressing generality is needed.

Program Safety Without Garbage Collection: All the previous applications of pool allocation focus on improving performance. Another major application of pool allocation has been to enforce program safety while allowing explicit memory deallocation for C progams (the techniques for a type-safe subset of C are described in [19] and a major extension to nearly arbitrary C is in progress). This work exploits two key properties: pools in type-safe programs are type homogeneous, and the segregation of individual data structures into pools ensures that many pools are relatively short-lived. The type-homogeneity means that even with explicit deallocation, we can prevent dangling pointers into the pool from being able to cause unintended type violations. The short pool lifetimes ensure that memory consumption does not increase significantly.

8. Implementation

We implemented Automatic Pool Allocation as a link-time transformation using the LLVM Compiler Infrastructure [34]. Performing cross-module, interprocedural techniques (like automatic pool allocation) at link-time has two advantages [3]: it preserves most of the benefits of separate compilation (requiring few or no changes to Makefiles for many applications), and it ensures that as much of the program is available for interprocedural optimization as possible.

Our system compiles source programs into the LLVM representation (for C and C++, we use a modified version of the GCC front-end), applies standard intraprocedural optimizations to each module, links the LLVM object files into a single LLVM module, and then applies interprocedural optimizations. At this stage, we first compute the complete Bottom-up DS graphs and then apply the Pool Allocation algorithm in Figure 6. Finally, we run a few passes to clean up the resulting code, the most important of which are interprocedural constant propagation (IPCP), to propagate null or global pool descriptors when these are passed as function arguments, and dead argument elimination (to remove pool pointer arguments made dead by IPCP). The resulting code is compiled to either native or C code using one of the LLVM back-ends, and linked with any native code libraries (i.e., those not available in LLVM form) for execution.

Our implementation of Data Structure Analysis and Automatic Pool Allocation are publicly available from the LLVM web site (`http://llvm.cs.uiuc.edu/`), together with the LLVM infrastructure, front-ends for C and C++, and most of the benchmarks used in the experimental evaluation in Section 9.

9. Experimental Results

We evaluated Automatic Pool Allocation experimentally in order to study several issues: compilation time, overall performance impact of pool allocation, the contributions of the later optimizations it enables, and the effect on the performance of the memory hierarchy.

For our experiments in this paper, we used the LLVM-to-C back-end and compiled the resulting C code with GCC 3.4.2 at -O3. The experiments were run on an AMD Athlon MP 2100+ running Fedora Core 1 Linux. This machine has exclusive 64KB L1 and 256KB L2 data caches. The C library on this system implements `malloc/free` using a modified Lea allocator, which is a high quality general purpose allocator. This allocator is used in all our experiments below, either directly from the application or the pool runtime library. All runtimes reported are the minimum user+system time from three executions of the program.

For this work, we are most interested in heap intensive programs, particularly those that use recursive data structures. For this reason, we include numbers for the pointer-intensive SPECINT 2000 benchmarks, the Ptrdist suite [2], the Olden suite [39], and the FreeBench suite [40]. We also include a few standalone programs: Povray3.1 (a widely used open source ray tracer, available from `povray.org`), espresso, fpgrowth (a patent-protected, data mining algorithm [25]), llu-bench (a linked-list microbenchmark) [46], and "chomp" from the McGill benchmark suite. All but SPEC, fp-growth and povray31 are available from `llvm.cs.uiuc.edu`.

For lack of space, we elide many benchmarks from these suites that were unaffected by pool allocation. This happens for several reasons. Some of the benchmarks, including 181.mcf, 186.crafty, 256.bzip2, and several FreeBench benchmarks, have very few dynamic memory allocations. A few (e.g. 197.parser, 254.gap, 255.vortex) have custom memory allocators, which prevents disambigution of allocated memory objects and causes all objects to be placed in a single pool. As an experiment, we removed the custom memory allocator from 197.parser and replaced it with wrappers that just call `malloc/free`; this is called 197.parser-b below. We can do this to 197.parser (but not the others) because its custom allocator has semantics identical to `malloc/free`. Finally, almost all the codes in the McGill benchmark suite have run times that are too small to be measured reliably.

9.1 Pool Allocation Statistics

Table 1 shows several basic statistics about pool allocation for each program. The StatPools column shows the number of static pools created in the program (when using Selective PA). The NumTH column shows the static number of type homogenous pools, and TH% is percentage of static pools that are type-homogenous. The DynPools column lists the number of dynamic pools created by the program. Tot Args and Max Args are the total number of formal arguments added to the program across all functions, and the maximum number for a single function.

Program	LOC	Stat Pools	Num TH	TH%	Dyn Pools	Tot Args	Max Args
164.gzip	8616	4	4	100%	44	1	1
175.vpr	17728	107	91	85%	44	23	4
197.parser-b	11204	49	48	98%	6674	76	16
252.eon	35819	124	123	99%	66	549	41
300.twolf	20461	94	88	94%	227	1	1
anagram	650	4	3	75%	4	0	0
bc	7297	24	22	32%	19	6	2
ft	1803	3	3	100%	4	0	0
ks	782	3	3	100%	3	0	0
yacr2	3982	20	20	100%	83	0	0
analyzer	923	5	5	100%	8	0	0
neural	785	5	5	100%	93	0	0
pcompress2	903	5	5	100%	8	0	0
llu-bench	191	1	1	100%	2	0	0
chomp	424	4	4	100%	7	10	8
fpgrowth	634	6	6	100%	3.4M	10	6
espresso	14959	160	160	100%	100K	191	13
povray31	108273	46	5	11%	14	290	4
bh	2090	1	0	0%	1	0	0
bisort	350	1	1	100%	1	1	1
em3d	682	12	12	100%	12	3	2
health	508	2	2	100%	2	4	2
mst	432	4	4	100%	4	0	0
perimeter	484	1	1	100%	1	1	1
power	622	3	3	100%	3	9	7
treeadd	245	6	6	100%	6	1	1
tsp	579	1	1	100%	1	1	1

Table 1. Basic Pool Allocation Statistics

The programs vary greatly in terms of the ratio of dynamic pool instances (Dyn Pools) to static pools (Stat Pools). `fpgrowth` has a particularly high ratio because it creates a new pool (for a local search tree) in each call to a recursive function. The number of arguments added to the programs is generally modest. 252.eon has a large number of arguments added because the standard C++ library is statically linked in, providing a large amount of cold code.

The Th% column also shows that for most pools, DSA is able to successfully prove that memory in the pool is used in a type-consistent manner, which we have found true across a wide range of C programs. This allows intelligent alignment decisions, gives the pool runtime information about expected size for single objects, and enables other novel compiler techniques described briefly in Section 7.

Program	BP	BP%	PFE
164.gzip	1	25%	9
175.vpr	27	25%	29
197.parser-b	3	6%	0
252.eon	0	0%	28
300.twolf	61	65%	1
anagram	2	50%	0
bc	3	13%	0
ft	2	67%	0
ks	3	100%	0
yacr2	7	35%	0
analyzer	5	100%	0
neural	5	100%	0
pcompress2	0	0%	0

Program	BP	BP%	PFE
llu-bench	1	100%	0
chomp	0	0%	0
fpgrowth	0	0%	0
espresso	1	1%	3
povray31	6	13%	28
bh	1	100%	0
bisort	1	100%	0
em3d	6	50%	0
health	2	100%	0
mst	4	100%	0
perimeter	1	100%	0
power	3	100%	0
treeadd	2	33%	0
tsp	1	100%	0

Table 2. Statistics for Pool Optimizations

Table 2 shows the static number of pools that can use a bump pointer after poolfree elimination (BP), and number of `poolfree` calls deleted when PoolFree Elim is enabled (PFE). The table shows that in many programs (the larger such examples are vpr, twolf, yacr2, and povray), a significant fraction of pools are identified as eligible bump-pointer pools, i.e., individual pool objects are never freed back to the pool. For vpr, twolf and povray, this is enabled by the elimination of several `poolfree` operations. This elimination indicates the presence of the build-use-destroy pattern explained in Section 5. In 175.vpr, for example, pool allocation eliminates 29 poolfree calls.

9.2 Pool Allocation Compile Time

Table 3 shows the compile times for pool allocation on programs bigger than 1000 lines of code. It breaks down this time into three components: the total time for DSA (which can be used by other clients as well), the time to compute the EBU graphs described in Section 3.3.2 (which are specific to pool allocation), and the time to perform the pool allocation transformation itself. The GCC column lists the time to compile the program with GCC 3.4.2 at -O3.

The total compilation time for pool allocation is extremely modest, taking less than 1.25 seconds in all cases on our Athlon 2100+. The largest amount of time is spent analyzing 252.eon (which has a large portion of the standard C++ library statically linked into it), followed by povray31; these are the only programs that took more than 1 second. Furthermore, much of the time is spent in DSA, which can be used for a variety of applications besides pool allocation [32]. Our implementation of the EBU and PA passes have not been optimized substantially, so they could probably be further reduced. Overall, these compilation times are extremely small for a sophisticated interprocedural optimization.

Program	LOC	GCC	DSA	EBU	PA	Total	GCC%
164.gzip	8616	2.67	0.02	0.01	0.01	0.03	1.1%
175.vpr	17728	9.39	0.06	0.03	0.05	0.14	1.5%
197.parser-b	11204	9.03	0.08	0.05	0.05	0.18	1.9%
252.eon	35819	131.13	0.51	0.30	0.42	1.23	0.9%
300.twolf	20459	17.21	0.09	0.07	0.03	0.19	1.1%
bc	7297	3.55	0.03	0.02	0.01	0.06	1.7%
ft	1803	0.68	0.01	0.01	0.01	0.02	2.9%
yacr2	3982	1.79	0.02	0.01	0.01	0.03	1.7%
espresso	14959	10.28	0.14	0.08	0.08	0.30	2.9%
povray31	108273	39.20	0.58	0.33	0.27	1.18	3.0%
bh	2090	0.85	0.01	0.01	0.01	0.01	1.2%

Table 3. Compile time (seconds) for programs > 1000 LOC

To put these times in perspective, the GCC% column (computed as (Total/GCC)*100), shows that the pool allocation transformation takes 3% or less of the time taken by GCC to compile these programs. This is significant because GCC -O3 performs no cross-module optimizations and inlining is the only interprocedural optimization it performs within a module. Overall, we believe these compilation times are quite acceptable for a production compiler.

9.3 Baselines and Pool Allocation Overheads

Program	GCC	NoPA	One - Pool	OnePool Run %	Only - OH	OnlyOH Run %
164.gzip	25.11	28.16	28.44	101.0%	28.17	100.0%
175.vpr	10.54	10.88	10.86	99.8%	10.87	99.9%
197.parser-b	12.59	12.42	17.86	142.7%	13.36	106.7%
252.eon	1.15	0.86	0.85	98.8%	0.88	102.3%
300.twolf	20.26	20.10	19.98	99.4%	20.50	102.0%
anagram	3.46	3.02	3.01	99.7%	3.02	100.0%
bc	1.71	1.55	1.48	95.5%	1.71	110.3%
ft	63.74	68.73	66.08	96.1%	68.94	100.3%
ks	4.56	4.43	5.30	119.6%	4.39	99.1%
yacr2	3.76	3.86	3.94	102.0%	3.89	100.8%
analyzer	324.54	312.25	314.69	100.8%	313.69	100.5%
neural	88.82	87.34	87.35	100.0%	87.60	100.3%
pcompress2	38.61	37.77	37.44	99.1%	38.04	100.7%
llu-bench	106.63	106.50	108.86	102.2%	106.76	100.2%
chomp	17.26	16.71	10.63	63.6%	16.82	100.6%
fpgrowth	36.27	36.62	36.49	99.7%	39.30	107.3%
espresso	1.25	1.22	1.20	98.3%	1.26	103.3%
povray31	9.41	9.79	9.69	98.9%	9.81	100.2%
bh	14.02	9.33	9.32	99.9%	9.35	100.2%
bisort	12.59	13.06	13.14	100.6%	13.20	101.1%
em3d	9.55	6.80	6.76	99.4%	6.80	100.0%
health	14.11	13.99	13.39	95.7%	13.98	99.9%
mst	12.79	13.14	13.23	100.7%	13.34	101.5%
perimeter	3.02	2.92	2.58	88.4%	3.00	102.7%
power	4.61	2.91	2.93	100.7%	2.92	100.3%
treeadd	17.48	17.41	17.29	99.3%	17.6	101.1%
tsp	7.17	7.24	7.08	97.8%	7.42	102.5%

Table 4. Baseline (NoPA), allocator, and overhead comparisons

Table 4 shows data to characterize the baseline we use for comparison and isolate the overheads added to a program by pool allocation. The GCC column is the execution time of the program with the GCC 3.4.2 compiler (at -O3). The NoPA column is the program compiled with LLVM using exactly the same sequence of transformation and cleanup passes as we do for pool allocation (see Section 8), but with the pool allocator and all pool-based optimizations disabled. Using NoPA as a baseline for comparison below isolates the speedup of the pool allocator transformation and its optimizations by factoring out the impact of other LLVM compiler passes. Comparing GCC to NoPA shows that the LLVM-generated code is no worse than 12% slower than GCC code and is sometimes much better. This indicates that the code quality of NoPA is reasonable to use a baseline for comparisons.

Another key question is how the difference between the allocator in our pool runtime library (used after pool allocation) and the standard libc malloc library (used by NoPA) affect the comparisons. This is significant because our pool library implementation is currently not thread-safe (though it is otherwise fully general), and this or other implementation details could skew the results in our favor. To measure this, we transformed the programs to allocate out of a single global pool (this transformation does not add pool arguments or other overhead to the program), effectively using our allocator to replace malloc and free for the program (the OnePool column). Comparing with NoPA shows that in all but 4 cases (197.parser-b, ks, chomp and perimter), OnePool is within about 5% of NoPA. The large slowdown for parser-b occurs because we use a singly-linked free list and the order of frees prevents coalescing adjacent free blocks. `chomp` is much faster with our allocator because our allocator has a fast path for fixed size allocations (to exploit type homogeneous pools) and nearly all allocations in `chomp` are (multiples of) this fixed size. As shown below, in all cases except perimeter, any such advantages from our runtime library (even chomp) are much smaller than the aggregate performance improvements due to pool allocation.

Finally, the OnlyOH column aims to isolate the performance overheads in the transformed code, namely, extra pool arguments on functions and initializing and destroying pool descriptors. It is computed by pool-allocating the program, but modifying the runtime library so that poolalloc/free simply call malloc/free. Comparing to NoPA shows that this overhead is negligible or quite low (less than about 5%) in nearly all cases, but is slightly higher in 197.parser-b (7%), bc(10%), and fpgrowth (7%). The pool allocator must overcome this overhead to provide a net performance improvement.

9.4 Pool Allocation and FullPA Aggregate Performance

Table 5 shows the program running time and speedups (relative to NoPA) for automatic pool allocation alone (BasePA) and for pool allocation with all pool-based optimizations (FullPA). FullPA therefore represents the aggregate performance impact of this work. As the table shows, FullPA improves the performance of many programs from 5% to 20%, improves analyzer and llu-bench by roughly 2x, and ft and chomp more than 10x. In no case does FullPA hurt the performance of other programs relative to NoPA. Not surprisingly, there is no obvious correlation between the speedups obtained and the number of static or dynamic pools. The causes and breakdown of these improvements are studied below.

Program	NoPA	BasePA	BasePA/ NoPA	FullPA	FullPA/ NoPA
164.gzip	28.09	27.93	0.99	28.40	1.01
175.vpr	10.88	10.85	1.00	10.30	0.94
197.parser-b	12.52	10.14	0.81	9.84	0.79
252.eon	0.86	0.84	0.98	0.84	0.98
300.twolf	20.10	17.59	0.88	17.01	0.85
anagram	3.02	3.00	0.99	3.00	0.99
bc	1.55	1.26	0.81	1.24	0.80
ft	68.73	5.89	0.09	4.98	0.07
ks	4.43	4.38	0.99	4.39	0.99
yacr2	3.89	3.89	1.01	3.87	1.00
analyzer	312.25	183.64	0.59	130.53	0.42
neural	87.60	87.33	1.00	87.15	1.00
pcompress2	38.04	37.52	0.99	37.68	1.00
llu-bench	106.50	108.37	1.02	60.96	0.57
chomp	16.71	1.71	0.10	1.46	0.09
fpgrowth	36.62	31.13	0.85	30.42	0.83
espresso	1.22	1.15	0.94	1.09	0.89
povray31	9.79	9.31	0.95	9.12	0.93
bh	9.33	9.41	1.01	8.88	0.95
bisort	13.06	13.02	1.00	11.04	0.85
em3d	6.80	6.82	1.00	6.62	0.97
health	13.99	13.35	0.95	12.02	0.86
mst	13.14	11.67	0.89	11.39	0.87
perimeter	2.92	2.59	0.89	2.45	0.84
power	2.91	2.91	1.00	2.91	1.00
treeadd	17.41	17.19	0.99	16.85	0.97
tsp	7.24	7.03	0.97	5.95	0.82

Table 5. Run time (seconds) and runtime ratios vs. NoPA

9.5 Locality improvements

Figure 10 shows the measured cache miss ratio of FullPA compared to NoPA, for each program that sped up at least 5%. The runtime ratios for these programs are shown in Figure 9 to help correlate the improvements in cache misses and running times. The figure includes data for the number of accesses that miss the Athlon's L1 D-cache, the number of accesses that miss the L2 D-cache, and the number of DTLB misses as measured by the Athlon performance monitoring counters. The graph shows that the programs with the largest speedups generally have dramatically reduced miss rates at every level of the cache hierarchy. The benefits for `twolf` and `llu-bench` are primarily at the TLB and those of `ft` are much greater at the cache. For all other cases, the reductions are closely

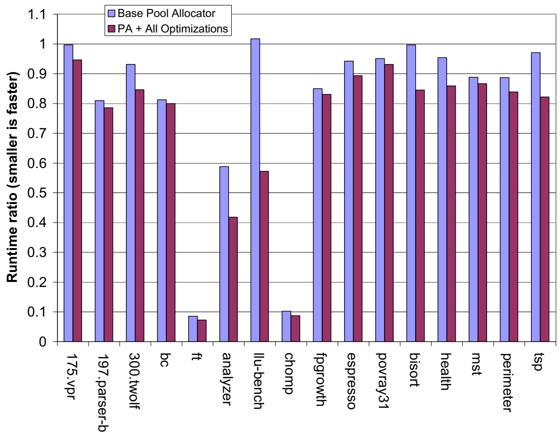


Figure 9. Aggregate execution time ratios (1.0 = NoPA)

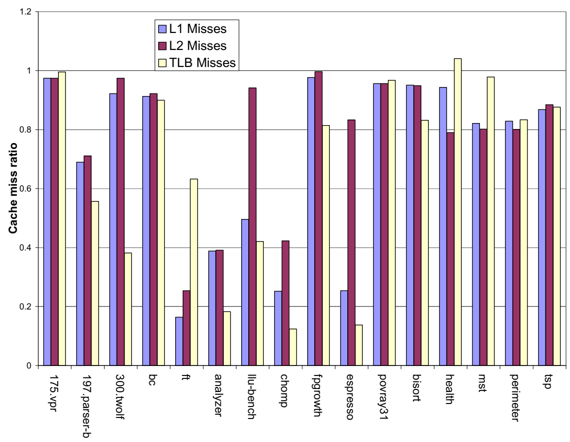


Figure 10. Ratio of misses at L1/L2/TLB: FullPA/NoPA

correlated at all the three levels of the memory hierarchy. This indicates that in these cases, the performance benefits are primarily due to smaller working sets, which would be produced by defragmenting the heap.

To characterize this effect, we discuss chomp in more detail. Chomp allocates three different nodes, which we call L, P, & D. L is an 8-byte object of type `_list`, P is a 16-byte object of type `_play`, and D is an array of int. Chomp uses an irregular allocation pattern, but generally intermixes object allocations (e.g. it starts with DPDLDPDLDLDDDPDLDL...). When using malloc, these objects are interspersed on the heap, roughly corresponding to allocation order (reuse of freed memory makes it inexact). When using the pool allocator, the three different objects are put in separate pools, and objects in each pool are roughly in allocation order (P is exactly in allocation order).

These layout patterns mean that, without Pool Allocation, the L and P list nodes are dispersed in memory (e.g. with variable strides of 100-500 bytes for the P objects) whereas the pool allocator packs them together (achieving a perfect stride of 20 bytes for the P objects, 16 for the object and 4 for the object header). This change dramatically reduces the cache footprint of linked list traversals over the P and L nodes. In the case of the P list, it yields optimal cache density and provides the hardware stride prefetcher with a linear access pattern. This combination provides a reduction from 251M L1 misses to 63M L1 misses. While chomp is an extreme case, it illustrates exactly the effect we aim for.

9.6 Contributions of Individual Optimizations

Figure 11 shows the runtime ratio of each program with one optimization disabled at a time, and compares it to a baseline of all optimizations on. This shows how much the program slows down when a particular optimization is disabled, which is correlated to how much the optimization helps the performance of the code. Note that if two optimizations can provide the speedup (e.g. either use of alignment-opt or bump-pointer to reduce inter-object padding), disabling either will not show a slowdown. Despite this, this analysis does provide useful insight into the effect of the optimizations.

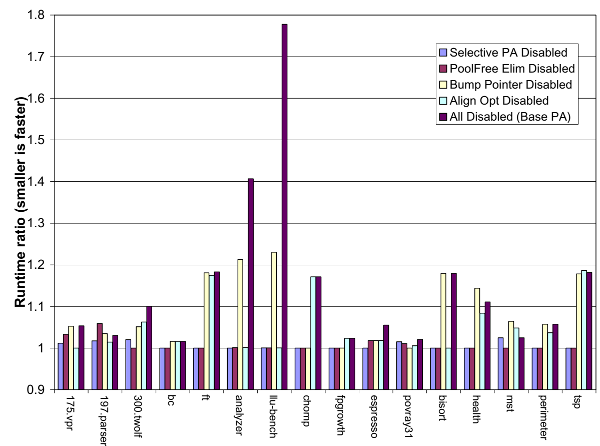


Figure 11. Optimization contributions (1.0 = FullPA, all optzns)

All of the optimizations except SelectivePA contribute noticeable improvements to at least one program. SelectivePA provides no significant speedup but does not hurt performance and it is useful because it can improve memory consumption significantly in some cases. The poolfree optimization improves 175.vpr, 197.parser-b, espresso, and povray31. The bump pointer optimization appears to be the most significant of the three, being particularly valuable to 175.vpr, 300.twolf, ft, analyzer, llu-bench, and several Olden programs. Close inspection of 175.vpr is particularly interesting: BasePA is not faster than NoPA, but a combination of poolfree elimination and the bump pointer optimization reduces the runtime of the program to 95.7% of NoPA (SelectivePA reduces it further to 94.6%). Finally, several programs benefited from the alignment optimization, particularly ft, chomp, health and tsp.

The speedup potential of these simple pool optimizations are particularly notable because they are all very simple optimizations, but can only be performed only once the heap has been segregated into pools.

10. Related Work

The primary goal of the pool allocation transformation is to give the compiler some control over the layout of data structures in the heap. We achieve this using a context-sensitive points-to graph to distinguish data structure instances and object lifetimes. We first contrast this work with previous approaches for influencing the layout of heap objects, and then with previous work on partitioning the heap for automatic (region-based) memory management.

Chilimbi et al. [12] describe a semi-automatic tool called `ccmorph` that reorganizes the layout of homogeneous trees at runtime to improve locality. It relies on programmer annotations to identify the root of a tree and to indicate the reorganization is safe. We automatically identify and segregate instances of many kinds of logical data structures, but do not yet identify when a *runtime* reorganization would be safe. They also describe another tool, ccmalloc, which is a malloc replacement that accepts hints to allocate one object near another object. These hints only provides local information for an object pair and not any global information about entire data structures.

Hirzel et al. [28] describe a technique to improve the effectiveness of Garbage Collection by partitioning heap objects according to their connectivity properties. Unlike our work, their partitions are not segregated on the runtime heap, are not directly related to distinct data structures, and the graph of partitions is restricted to be a DAG, which prevents fine grained partitioning of mutually recursive structures (like graphs).

Several proposed techniques aim to improve storage allocation or GC performance by relating objects based on their predicted lifetimes [26, 18, 4, 15, 13, 41]. These techniques use heuristics such as allocation site, call stack, or object size, combined with profiling information, to predict lifetime properties approximately. In contrast, our approach uses a more rigorous analysis to group objects both by structural relationships and statically derived lifetimes.

Other authors have developed techniques (usually profile-based) to reorganize fields within a single structure or place objects near each other to improve locality of reference [23, 9, 41, 11, 29]. These placement decisions are orthogonal to the choices made by Automatic Pool Allocation, and could therefore be combined with our transformation. This an important direction for future work.

There has been significant work on runtime libraries for region-based memory management [5], and on language mechanisms for manual region-based memory management as an alternative to garbage collection, e.g., Real-time Java [7], RC [21], Cyclone [30, 22], and others [21, 17, 8]. Compared with our approach, these library- or language-based techniques are much easier to implement, but require significant manual effort to use. In addition, although the region-based libraries and languages expose the relationship between objects and regions to the compiler, they do not expose any notion of higher-level data structures or how they relate to objects and regions. Therefore, the compiler does not obtain information about data structures and traversals that could enable optimizations on logical data structures.

There is a rich body of work on *automatic* region inference as a technique for memory management, for both functional [45, 44, 1, 24] and object-oriented languages [14, 10]. Unlike this body of our work, our primary goal is to segregate and control the layout of data structures in the heap for better performance and to enable subsequent compiler techniques that exploit knowledge of these layouts. We describe several optimizations that exploit data structure pools, and explore the performance implications of data structure segregation on program performance in some detail. There are also some key technical differences between this prior work and ours. First, all these previous techniques except the work of Cherem and Rugina [10] are based on type inference with a region-based type system. It does not appear straightforward to extend the type inference approaches to work for weakly-typed languages like C and C++, which can contain pointer casts, varargs functions, unions, etc., on which type information is difficult to propagate statically. In contrast, both our underlying pointer analysis and our transformation algorithm correctly handle all the complex features of C and C++, by distinguishing objects with known and unknown type (in the points-to graph) and by using a conservative and very efficient graph merging technique (the same as in DSA) to deal with potentially type-unsafe uses of pointers during the transformation. Second, using a pointer analysis as the basis for our transformation enables additional optimizations by exploiting the explicit relationship between a points-to graph and pools. Finally, the use of type inference and a rich type-system is not well suited for modern optimizing compilers, which are usually based on a mid-level or low-level internal representation supporting multiple source languages. Our approach is specifically designed for use in such compilers, and relies only a simple, mid-level intermediate representation and pointer analysis.

The work of Cherem and Rugina [10] was performed concurrently with ours and our approaches are technically similar in some key ways. They describe a region inference approach for Java based on a flow-insensitive, context-sensitive points-to analysis. Because their primary focus is automatic memory management, they are much more aggressive about computing region lifetimes, including loop-carried regions. Our regions *can* be placed as flexibly as theirs, but we use a simpler placement analysis. Like the type-inference approaches, however, their work also does not support weakly typed languages like C. Although the underlying pointer analysis could be extended to do so (using our approach, for example), we believe the transformation would be more difficult to extend. Furthermore, they too focus on automatic memory management, and do not explore the impact of their work on memory hierarchy performance or consider other optimizations that could exploit their region information. We expect that our optimization techniques could be fruitfully combined with their region inference algorithm for Java programs.

There is a wide range of work on techniques for stack allocation of heap objects as well as techniques for static garbage collection, both of which are based on analyzing the lifetimes of objects in programs (e.g., see [6, 42, 31] and the references therein). These techniques do not attempt to analyze or control the layout of logical data structures in the heap *per se*, and are largely orthogonal to our work. A minor exception is that our optimization to eliminate poolfree for a pool (when there are no intervening allocations before the subsequent pooldestroy) essentially replaces explicit deallocation with static reclamation of memory in the pool. This is the inverse of (and much more limited than) the work on static GC, which aims to replace or optimize runtime GC.

Finally, in an early workshop paper [33], we proposed the basic idea of Automatic Pool Allocation. That work did not consider how to handle the key difficult cases for this transformation, namely, function pointers and efficient handling of recursion. It relied on an early, non-scalable version of DSA (which did not support practical handling of non-type-safe data structures), did not describe any optimizations to exploit pool allocation, and did not evaluate the performance impact of pool allocation. The current algorithm is general, practical and efficient, and supersedes the previous work.

11. Conclusions and Future Work

The primary contribution of this paper is a practical, efficient compiler algorithm to segregate distinct instances of logical data structures into separate pools in the heap. Our implementation of the algorithm applies to the full generality of C and C++ programs and performs several additional optimizations that take advantage of pool allocation. Our results show that for many programs, the transformation achieves the major goal stated in the Introduction, namely, that it can improve program performance, sometimes quite substantially. The complete implementation and most of our benchmarks are publicly available at llvm.cs.uiuc.edu.

We believe that the combination of Data Structure Analysis and Automatic Pool Allocation together provide a new foundation for analyzing and transforming pointer-intensive programs, not in terms of individual memory references or data elements but rather *in terms of how such programs create and use entire logical data structures*. We term this a "macroscopic" approach to pointer-intensive data structures. The broad goal of our ongoing work is to continue investigating novel macroscopic techniques for program optimization, program monitoring, memory safety, and automatic memory management. The first two major examples described briefly in Section 7 illustrate the potential power of this approach for enabling novel solutions to difficult compiler problems.

Acknowledgments

This work has been supported in part by an NSF CAREER Award (EIA-00-93426), the NSF Next Generation Software Program (EIA-01-03756), the MARCO Focus Research Center Program through GSRC, and by an Intel Graduate Fellowship. The authors would like to thank John Criswell for his assistance with Linux performance monitoring counters and with several benchmarks, and Sumant Kowshik for his contributions to parts of the implementation. We are also grateful to Shengnan Cong for her assistance with the fpgrowth program, the members of the LLVM group for their comments and suggestions on the paper, and the anonymous reviewers for their valuable feedback.

References

[1] A. Aiken, M. Fähndrich, and R. Levien. Better static memory management: Improving region-based analysis of higher-order languages. In *PLDI*, pages 174–185, June 1995.

[2] T. Austin, et al. The Pointer-intensive Benchmark Suite. `www.cs.wisc.edu/~austin/ptr-dist.html`, Sept 1995.

[3] A. Ayers, S. de Jong, J. Peyton, and R. Schooler. Scalable cross-module optimization. In *PLDI*, Montreal, June 1998.

[4] D. A. Barrett and B. G. Zorn. Using lifetime predictors to improve memory allocation performance. In *PLDI*, pages 187–196, Albuquerque, New Mexixo, June 1993.

[5] E. D. Berger, B. G. Zorn, and K. S. McKinley. Reconsidering custom memory allocation. In *OOPSLA*, Seattle, Washington, Nov. 2002.

[6] B. Blanchet. Escape Analysis for Java(TM): Theory and Practice. *TOPLAS*, 25(6):713–775, Nov 2003.

[7] G. Bollella and J. Gosling. The real-time specification for Java. *Computer*, 33(6):47–54, 2000.

[8] C. Boyapati, A. Salcianu, W. Beebee, and M. Rinard. Ownership types for safe region-based memory management in real-time java. In *PLDI*, 2003.

[9] B. Calder, K. Chandra, S. John, and T. Austin. Cache-conscious data placement. In *Proc. ASPLOS-VIII*, pages 139–149, San Jose, USA, 1998.

[10] S. Cherem and R. Rugina. Region analysis and transformation for java programs. In *2004 Int'l Symposium On Memory Management*, Vancouver, Canada, Oct. 2004.

[11] T. M. Chilimbi, B. Davidson, and J. R. Larus. Cache-conscious structure definition. In *PLDI'99*, pages 13–24. ACM Press, 1999.

[12] T. M. Chilimbi, M. D. Hill, and J. R. Larus. Cache-conscious structure layout. In *PLDI'99*, pages 1–12. ACM Press, 1999.

[13] T. M. Chilimbi and J. R. Larus. Using generational garbage collection to implement cache-conscious data placement. *ACM SIGPLAN Notices*, 34(3):37–48, 1999.

[14] W.-N. Chin, F. Craciun, S. Qin, and M. Rinard. Region inference for an object-oriented language. In *PLDI*, Washington, DC, June 2004.

[15] R. Courts. Improving locality of reference in a garbage-collecting memory management system. *CACM*, 31(9):1128–1138, 1988.

[16] M. Das. Unification-based pointer analysis with directional assignments. In *PLDI*, pages 35–46, 2000.

[17] R. DeLine and M. Fähndrich. Enforcing high-level protocols in low-level software. In *PLDI*, Snowbird, UT, June 2001.

[18] A. Demers, M. Weiser, B. Hayes, H. Boehm, D. Bobrow, and S. Shenker. Combining generational and conservative garbage collection: framework and implementations. In *Proc. ACM POPL*, pages 261–269, 1990.

[19] D. Dhurjati, S. Kowshik, V. Adve, and C. Lattner. Memory safety without garbage collection for embedded applications. *Transactions on Embedded Computing Systems*, 4(1):73–111, Feb. 2005.

[20] M. Fähndrich, J. Rehof, and M. Das. Scalable context-sensitive flow analysis using instantiation constraints. In *PLDI*, Vancouver, Canada, June 2000.

[21] D. Gay and A. Aiken. Memory management with explicit regions. In *PLDI*, pages 313–323, Montreal, Canada, 1998.

[22] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in cyclone. In *PLDI*, June 2002.

[23] D. Grunwald and B. Zorn. Customalloc: Efficient synthesized memory allocators. *SP&E*, 23(8):851–869, 1993.

[24] N. Hallenberg, M. Elsman, and M. Tofte. Combining region inference and garbage collection. In *PLDI*, Berlin, Germany, June 2002.

[25] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *SIGMOD*, pages 1–12, 2000.

[26] D. R. Hanson. Fast allocation and deallcoation of memory based on object lifetimes. *SP&E*, 20(1):5–12, Jan 1990.

[27] M. Hind. Pointer analysis: haven't we solved this problem yet? In *PASTE*, pages 54–61. ACM Press, 2001.

[28] M. Hirzel, A. Diwan, and M. Hertz. Connectivity-based garbage collection. In *OOPSLA*, pages 359–373, 2003.

[29] X. Huang, S. Blackburn, K. McKinley, E. Moss, Z. Wang, and P. Cheng. The garbage collection advantage: improving program locality. In *OOPSLA*, pages 69–80, 2004.

[30] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *USENIX Annual Technical Conference*, Monterey, CA, 2002.

[31] R. Jones. *Garbage Collection. Algorithms for Automatic Dynamic Memory Management*. John Wiley & Sons, 1999.

[32] C. Lattner. *Macroscopic Data Structure Analysis and Optimization*. PhD thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, May 2005. *See* `http://llvm.cs.uiuc.edu`.

[33] C. Lattner and V. Adve. Automatic Pool Allocation for Disjoint Data Structures. In *MSP*, Berlin, Germany, Jun 2002.

[34] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. In *CGO*, San Jose, USA, Mar 2004.

[35] C. Lattner and V. Adve. Transparent Pointer Compression for Linked Data Structures. In *Proc. ACM Workshop on Memory System Performance*, Chicago, IL, Jun 2005.

[36] D. Liang and M. J. Harrold. Efficient points-to analysis for whole-program analysis. In *ESEC / SIGSOFT FSE*, pages 199–215, 1999.

[37] D. Liang and M. J. Harrold. Efficient computation of parameterized pointer information for interprocedural analysis. In *SAS*, July 2001.

[38] E. M. Nystrom, H.-S. Kim, and W. mei W. Hwu. Bottom-up and top-down context-sensitive summary-based pointer analysis. In *SAS*, 2004.

[39] A. Rogers, M. Carlisle, J. Reppy, and L. Hendren. Supporting dynamic data structures on distributed memory machines. *TOPLAS*, 17(2), Mar. 1995.

[40] P. Rundberg and F. Warg. The FreeBench v1.0 Benchmark Suite. `http://www.freebench.org`, Jan 2002.

[41] M. L. Seidl and B. G. Zorn. Segregating heap objects by reference behavior and lifetime. In *ASPLOS-VIII*, pages 12–23, San Jose, USA, 1998.

[42] R. Shaham, E. Yahav, E. K. Kolodner, and M. Sagiv. Establishing local temporal heap safety properties with applications to compile-time memory management. In *SAS*, San Diego, USA, June 2003.

[43] B. Steensgaard. Points-to analysis in almost linear time. In *POPL*, pages 32–41, Jan 1996.

[44] M. Tofte and L. Birkedal. A region inference algorithm. *TOPLAS*, 20(4):724–768, July 1998.

[45] M. Tofte and J.-P. Talpin. Implementation of the typed call-by-value λ-calculus using a stack of regions. In *POPL*, pages 188–201, 1994.

[46] C. B. Zilles. Benchmark health considered harmful. *SIGARCH Comput. Archit. News*, 29(3):4–5, 2001.