Automatic Pool Allocation: Compile-Time Control of Data Structure Layout in the Heap

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

Abstract

Despite the potential importance of data structure layouts and traversal patterns, compiler transformations on pointer-intensive programs are performed primarily using pointer analysis, and not by controlling and using information about the layout of high-level data structures. This paper describes a compiler transformation called *Automatic Pool Allocation* that segregates instances of "logical" data structures in the heap into distinct pools, and allows different heuristics to be used to partially control the internal layout of those data structures. Because these are rigorous transformations, their results, combined with pointer analysis information, can be used to perform further compiler analyses and transformations, and we briefly list a few examples. Automatic Pool Allocation also provides several direct performance benefits for pointer intensive programs, most importantly, that traversals of a logical data structure allocated to a separate pool can have better spatial locality and smaller working sets. We evaluate the performance and cache behavior of the code transformed by the automatic pool allocation transformation on a series of heap-intensive and general-purpose benchmarks, and find that it speeds up several C programs by 10-40% percent or more, and does not hurt (or help) other programs.

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, but have much less information about complex data structures allocated on the heap. For pointer-based data structures, however, both the relative layout of *distinct data structures* in memory and the 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 is that many compiler techniques (e.g., software prefetching, data layout transformations, and safety analysis) are either less effective or inapplicable for complex data structures.

Despite the potential importance of data structure layouts and traversal patterns, compiler transformations on pointer-intensive programs are performed based primarily on pointer and dependence analysis, and not *by controlling and using information about the layout of high-level data structures.* There are a few compiler techniques that modify the layout of higher-level pointer-based data structures (not just fields within a structure) in imperative programs, but none of them attempts to provide layout information to enable further compiler transformations, or can be directly extended to achieve this goal. This is because these techniques focus on giving hints or memory layout directives to a runtime library [12], memory

```
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

```
struct list { list *Next; int *Data; };
list* createnode(pooldesc *PD, int *Data) {
  list *New = poolalloc(PD, sizeof(list), 1);
  New->Data = Data;
  return New;
}
void splitclone(pooldesc *PD1, pooldesc *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;
  pooldesc PD1, PD2;  // initialize pools
  poolcreate(&PD1); poolcreate(&PD2);
  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**

This function copies a list into two disjoint lists A and B (based on some predicate), processes each, then frees them. After basic pool allocation, the new lists are allocated in separate pools (PD1 and PD2) and each is contiguous in memory. After two further optimizations, the calls to `poolfree` *for A and B are eliminated, as well as the loops containing them because the memory for both pools will be reclaimed by* `pooldestroy` *before* any *more memory is allocated.*

allocator [9], or garbage collector [23]. Furthermore, they do not analyze higher-level data structures in order to segregate them on the heap or control their internal layout at compile-time.

The work of Tofte, Birkedal et al. on automatic region inference for ML [37, 36] is a systematic compile-time transformation and could actually be used to enable subsequent compiler optimizations, although they do not propose to do so. Their work is aimed at using region inference for automatic memory management with little or no garbage collection, as they do in the ML Kit [36, 5, 21]. As we discuss in Section 8, the two major differences between their work and ours is that their compile-time transformation does not explicitly identify and control the internal layout of complex data structures (although they may put each instance in a separate region), and they do not use an explicit pointer analysis to track references in the presence of destructive heap updates, which is crucial for imperative languages like C, C++ and Java. The latter issue and other minor differences are discussed further in Section 8. Nevertheless, their transformation could be modified to enable further optimizations for the functional subset of ML as our work aims to enable for imperative languages.

The key insight underlying our work is that (any) context-sensitive, field-sensitive points-to analysis can be used to distinguish *disjoint instances of logical data structures* in a program, and identify the locations at which nodes of those data structures are created, accessed, and destroyed (these terms

are defined more precisely in Section 3). This gives the compiler the information needed to segregate individual data structure instances and to better control their internal layout. 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.

1.1 Contributions of this Work

This paper describes *Automatic Pool Allocation*, an automatic transformation framework for imperative programs that *segregates instances of "logical" data structures in the heap into distinct 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. Because these are rigorous transformations, their results, combined with pointer analysis information, can be used to perform further compiler analyses and transformations. We briefly describe several novel applications of our work in Section 9 that exploit this layout information and control, including techniques for enforcing heap safety by exploiting type homogeneous pools, replacing (large) pointers with (small) indices into a pool to reduce cache footprint of large pointer-based data structures, and more speculative ideas on pointer prefetching and traversal order transformations. Detailed descriptions or evaluation of these techniques are outside the scope of this paper.

Automatic Pool Allocation, with appropriate layout heuristics, can itself 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. 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 the processed in preorder), improving spatial locality and (if objects are smaller than a cache line) cache bandwidth. Third, in such cases the traversal order may become a simple linear stride, allowing effective hardware prefetching. Fourth, some pools hold objects of only a single type (and so a single size), speeding up allocation and deallocation. Finally, 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).

Figure 1 illustrates the pool allocation transformation for an example C program fragment. Given a list L, the code copies its nodes into two separate new lists A and B, processes each of the new lists, and then destroys them. The code on the right 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 become segregated in the heap, unlike the original program where they are likely to be interleaved in data-dependent fashion in memory. The compiler "knows" which pool contains which list, and some properties of the layout within each pool. 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 have implemented Automatic Pool Allocation as a link-time transformation using the LLVM Compiler Infrastructure [27]. Our interprocedural transformations are performed at link-time to preserve separate compilation. The implementation supports arbitrary C programs, including programs with function pointers, recursion, non-type-safe memory accesses, and incomplete programs. We evaluate the performance impact of Automatic Pool Allocation, and study the performance effects of the transformation on three suites of benchmarks. We find that several programs speed up by 10-40%, and one up by 99and importantly, are not hurt by the transformation.

2 Background: Points-To Graph Assumptions

In this section, we specify formally the points-to graph representation and properties that a pointer analysis must provide in order to support Automatic Pool Allocation. We know of at least three such

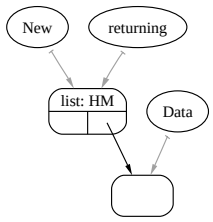

Figure 2: (a): BU Graph for createnode

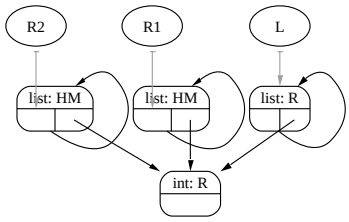

Figure 2(b): BU Graph for splitclone

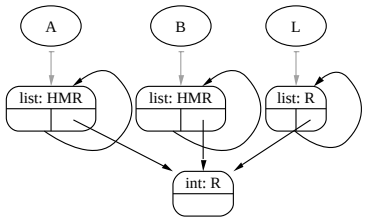

Figure 2(c): BU Graph for processlist

algorithms including our own [18, 28, 26] that provide or can be extended to generate graphs with these properties.

We use the term Data Structure Graph (DS Graph) to refer to the points-to graphs used by Automatic Pool Allocation. A DS graph is a compile-time description of the memory objects created by a program, and their points-to properties. Different nodes within the same graph represent distinct memory objects. We assume that a DS graph is computed for each function (of course, the same graph can be used for all functions at some loss of precision), representing the memory objects reachable from variables in that function or global variables.

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 type, or $\perp$ representing an unknown type. In the analysis, $\perp$ is treated like an unknown-size array of bytes. F is an array of fields, one for each possible field of the type τ. Scalar or array types have a single field. 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. A **U** node 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. Finally, A is a boolean that is true if the node includes an **A** rray object.

DS Edge : A DS edge is a 4-tuple: $\{s, f_s, t, f_t\}$, s, t are DS nodes, and 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 have no outgoing DS edge only if the field is known not to contain a pointer type, e.g., it is a function, floating point, or small integer type, or if $M = \{\mathbf{U}\}$.

Figure 2(b) shows the DS graph computed for function `splitclone` of our example program, computed by our compiler. Note that each node of type `list` has two fields. The cycles indicate recursive data structures. $R1$ and $R2$ point to distinct nodes, indicating that the two linked lists are completely disjoint. This allows pool allocation to put these two lists in distinct pools, even though they are created in interleaved fashion and by calling the same function.

Our algorithm that computes DS graphs is called Data Structure Analysis (DSA) [26]. DSA has several properties required for supporting Automatic Pool Allocation, which are also shared by the other two analyses mentioned above [18, 28] (which we refer to as FRD and LH). Most importantly, all three are *context-sensitive*, which is important for distinguish data structure instances that may be created, processed, and destroyed by calling common functions. For example, context-sensitivity allows DSA to determine that lists A and B are disjoint in Figure 1(a). DSA and LH distinguish objects by complete acyclic call paths, while FRD provides a more limited form of context-sensitivity. Pool allocation could still work with less or even no context-sensitivity (e.g., if heap objects were distinguished only by allocation site), but instances of a data structure processed by common functions would likely not be segregated into distinct pools.

All three algorithms are also *field-sensitive* (they distinguish distinct pointer fields within objects). LH and DSA are *unification-style* algorithms [34] while FRD uses inclusion constraints. We believe field-sensitivity is crucial for unification-style algorithms to support pool allocation because merging all targets of all pointer fields in a single object would lose most structural information about pointer-intensive data

structures. A non-unification based algorithm that distinguishes objects of different types may be able to support Automatic Pool Allocation effectively without requiring field-sensitivity [2]. In practice, however, we have found the "unification" property to be crucial for allowing context-sensitivity by complete acyclic call paths to be fast and to keep the graphs small [26] (in fact, imprecision tends to reduce the size of the graphs, due to merging of nodes).

Finally, the DS graph of a function must include the side-effects of any function reachable from the current one in the call graph (i.e., immediate callees and their callees and so on). These side-effects include all memory objects created or accessed that are reachable from variables in the current function, and all points-to relationships introduced on those objects. More specifically, for each node n in the callee's graph that is reachable in the caller, f, there is a unique node in the caller's graph corresponding to it which we denote $NodeInCaller(f, n)$. Both DSA and LH compute separate "bottom-up" and "top-down" graphs, and the bottom-up graphs provide exactly the information we need.

3 Logical Data Structures and Disjoint Instances

The high-level ("logical") data structures around which a program is designed are a subjective notion and difficult to formalize. The key property that we wish to exploit, however, is that pointer-intensive computations are usually designed as traversals over one or more connected data structure (arrays or pointer-based structures).

Based on this intuition, we define a *data structure snapshot* (DSS) to be a set of memory objects at runtime that form nodes of a connected graph, where an edge is formed by a pointer from one object element to another, and elements of individual objects can be traversed via structure or array indexing. This gives the key property that a DSS must be *traversable* via sequences of pointer dereferences and indexing operations. One DSS can contain another; for example, an array of binary trees forms a DSS, and so does each individual tree (viewed at a particular point in an execution).

The DS graph of the previous section can be viewed as a static description of the DSSs created by a program. Any DSS will be represented by some subgraph of the DS graph. Note that a single subgraph represents a large number of DSSs (for example, a linked list that grows and shrinks). Intuitively, a DS subgraph is a static description of a logical data structure, such that only one "instance" of this data structure can exist at any point in an execution. Disjoint subgraphs represent distinct data structures: more formally, DSSs of the two different subgraphs will not intersect.

4 The Core Pool Allocation Transformation

We first present a "core" version of the transformation in which all heap objects are allocated in pools (i.e., none are allocated directly via `malloc`) and every DS node generates a separate static pool (explained below). All steps of the algorithm consider only those DS nodes with $\mathbf{H} \in M$ (" $\mathbf{H}$ nodes") as candidates for allocating to pools. Because disjoint DS subgraphs generate disjoint logical data structure instances as defined above, this transformation automatically segregates such data structure instances in the heap. In the next section, we consider alternative heuristics that could be used for these two choices; the transformation steps only require one minor change to support arbitrary choices. All parts of the algorithm assume a call graph is available; in our implementation, it is computed as a side effect of DSA.

4.1 Overview Using an Example

We can first illustrate the basic steps of the transformation in intuitive terms using the two-lists example and its DS graphs in Figure 2.

First, from each function's DS graph, we determine which $\mathbf{H}$ nodes are accessible after their respective functions return, i.e., they "escape" to their callers. The $\mathbf{H}$ nodes in `createnode` and `splitclone` do

escape, whereas the the two in `processlist` (`A` and `B`) do not. The latter are candidates for creating new pools in `processlist`. For now, we decide that every such candidate should be assigned a pool, and each pool should be separate. In Section 5, we discuss alternative heuristics for each of these choices.

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 upwards for a function, i.e., for lists `R1` and `R2` in `splitclone` and for lists `New` in `createNode`. It rewrites the calls to `malloc` and `free` with calls to `poolalloc` and `poolfree`, passing the appropriate pool descriptors as arguments. Finally, it rewrites the calls to functions with pool descriptor arguments (i.e., the calls to `splitclone` and `createnode`), in order to pass in the right 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 and the loop enclosing those calls to free.

The final resulting code (not shown) has put each linked list into a separate pool on the heap, made the list objects of each list contiguous in memory, and reclaimed all the memory for each list at once instead of freeing items individually, without increasing the peak memory consumption of the program. In this example, the objects of a list are placed in dynamic creation order within each pool.

4.2 Pool Allocator Runtime Library

The interface to our pool allocation library is shown in Figure 3. The functions `poolalloc` and `poolfree` are used by the generated code to allocate and deallocate memory from a pool, i.e., these are used in place of calls to `malloc` and `free` in the program. Note that the third argument to `poolalloc` can be used to expose to the library that an array is being allocated with elements of a particular size. The function `poolcreate` initializes a pool descriptor for a new pool. `pooldestroy` releases any remaining memory in the specific pool to the system heap. 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) and returns blocks using `free` when they become empty. The allocation strategy is optimized for pools that have items of uniform size. In such a pool, requests for an item are always satisfied in constant time, when memory is available in the pool.

```
void poolcreate(Pool** PP);
    Creates and returns a new pool descriptor
void pooldestroy(Pool* PP)
    Release pool memory and destroys pool descriptor
void* poolalloc(Pool* PP, size_t sz, uint numElem)
    Allocate numElem objects of sz bytes each.
void poolfree (Pool* PP, void* ptr)
    Mark the object pointed to by ptr as free.
```

Figure 3: Interface to the Pool Allocator Runtime Library

4.3 Analysis: Finding Candidate Nodes for Pools

The goal of the analysis is to identify what pool descriptors must be available in each function, and which of those must be initialized within the function using `poolcreate` (the others will be passed in as arguments). We use the term *static pool* to refer to a single `poolcreate` statement in the generated code. During runtime, at most one dynamic instance of each static pool will exist at any time.

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 known callees. These identify exactly those nodes for which a pool descriptor must be *available* in the current function.

To determine which pool descriptors must be created within the current function, we must determine which DS nodes represent memory objects local to the function, and which ones represent objects used before or after the function exits. This is essentially an escape analysis [6], and can be done for all **H** nodes efficiently using the DS graph. In the DS graph, a DS node represents objects accessible outside the function if the node is reachable from the argument nodes for the function, return value nodes, or **G**lobal nodes. If none of these conditions is true, then the lifetime of the memory objects represented by that node are bounded by the lifetime of the function. All escaping DS nodes can be identified by a single, linear-time traversal of the DS graph (not shown).

4.4 Transformation: Creating and Passing Pool Descriptors

4.4.1 The Basic Transformation

We first describe the basic pool allocation transformation, ignoring indirect function calls. The pseudocode for this basic transformation, including the analysis steps, is shown in Figure 4. The algorithm makes two passes over the functions in the program in arbitrary order.

Lines 1–5 identify what DS nodes are candidates for inserting `poolcreate` statements within the function. For each other DS node with the **H** flag, the algorithm adds one pool descriptor pointer argument to the function (except for DS nodes pointed to by global variables, as described in section 4.5 below). We always add pool descriptor arguments to the start of a function's argument list in order to support variable-argument functions. The algorithm records the mapping of DS graph nodes to the pool arguments for each function, to use when rewriting the calls to the function.

For each node that needs a pool in the function, the algorithm allocates a pool descriptor on the stack frame, initializing it (using `poolinit`) on entry to the function and destroying the pool (`pooldestroy`) at every exit of the function (these placement choices are improved in Section **??**). 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 of was not deallocated by the original program. Note that this may actually eliminate some memory leaks in the program!

In the second pass (lines 11–20), the algorithm replaces calls to `malloc()` and `free()`[1] with calls to `poolalloc` and `poolfree`. We pass the pool descriptor pointer of the DS node pointed to by the pointer in each case, using the mapping information saved by the first pass. Since the DS node will have an **H** , a pool descriptor pointer is guaranteed to be available.

Calls to functions other than `malloc` or `free` must pass additional pool descriptor arguments for memory that escapes from them. Because the DS graph reflects all reachable memory objects of all 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, regardless of whether the callee is a recursive function or not. Identifying which pool of the caller (f) to pass for each pool argument of the callee is straightforward: for each callee node n that was recorded as needing an argument pool descriptor, we pass the pool descriptor for the node $NodeInCaller(f, n)$ in the caller's DS graph (defined in Section 2).

4.4.2 Passing Pool Descriptors for Indirect Function Calls

The presence of indirect function calls makes it significantly more complex to pass the correct pool descriptor arguments to each function. The problem is that different functions called via a function pointer at the same call site may require different sets of pools. Furthermore, different indirect call

[1] Note that "malloc wrappers" (like `calloc`, `realloc`, `valloc`, etc) do not need special support from the pool allocator. Their bodies are simply linked into the program and treated as if they were a user function, getting new pool descriptor arguments to indicate which pool to allocate from.

```
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           variable a = AddPoolDescArgument(f, n)
7           pdmap(n) = a                     // Remember pooldesc
8         else                               // Node is local to fn
9           variable pd = AddInitAndDestroyLocalPool(f, n)
10          pdmap(n) = pd
11  ∀f ∈ functions(P)
12    ∀i ∈ instructions(F)                   // Rewrite function
13      if (i isa 'Ptr = malloc(size)')
14        replace with 'poolalloc(pdmap(dsnode(Ptr)), size)'
15      else if (i isa 'free(Ptr)')
16        replace with 'poolfree(pdmap(dsnode(Ptr)))'
17      else if (i isa 'call Callee(args)')
18        ∀n ∈ poolDescArgNodes(Callee)
20          addCallArgument(pdmap(NodeInCaller(f, n)))
```

Figure 4: Pseudo code for basic algorithm

sites can have different but overlapping sets of potential callees. Our solution is shown in Figure 5, and described below.

The pool allocator uses the call graph to partition all the functions of the program into disjoint equivalence classes, such that any two functions callable from the same indirect call site will be in the same equivalence class. Initially, every function in the program is placed into its own equivalence class. For each indirect call site in the program, we unify the equivalence classes for all the potential callees at that site, using the Tarjan union-find algorithm.

Because our algorithm for matching pools between functions uses the DS nodes for the functions and the $NodeInCaller(f, n)$ map, the DS graph for a caller must reflect all the possible nodes that must be passed to any function in the equivalence class of a callee. To enforce this, the pool allocator merges all the graphs for all the functions in the equivalence class to build a single common DS graph[2]. We merge a pair of graphs in the equivalence class by merging corresponding formal argument nodes, global nodes, and the return value node of each graph.

```
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 5: Pseudo code for complete pool allocator

The last step is to incorporate the equivalence class graphs of the callees into the DS graph of each function so that the caller and "callee" graphs are consistent. We call these the EBU ("equivalence bottom-up") graphs. The EBU graph for a function is initially its equivalence class graph. We then traverse the SCCs of the call graph in bottom-up order using Tarjan's SCC finding algorithm. As in the original bottom-up phase of DSA, for each multiple-function SCC, we first collapse the EBU graphs for the functions of the SCC into a single graph. This reduces the call graph to a DAG with a single node for each original SCC. For each node in this DAG, we merge the completed EBU graph of each callee into the EBU graph of the caller. The end result of this process is that the EBU graph for each function represents the memory behavior of the function including any functions it can call indirectly and all functions in the same equivalence class (because it may not call some of them directly, the EBU

[2]Merging graphs is a primitive operation in our DSA implementation used for inlining graphs in the bottom-up pass.

graph is less precise than the original graph, but still correct).

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. At this point, we simply apply the algorithm of Figure 4 using the equivalence class graph instead of the original DS graph for each function. Because the graphs for all functions in an equivalence class are identical, all of their pool descriptor arguments will be identical. Also, when transforming the body of a function, the correct arguments would be added to an indirect call based on the graph for the called equivalence class.

4.5 Algorithm Refinements

There are 3 important refinements to the algorithm, but they are simple and we only describe them very briefly. The first is essential for efficiency, while the other two are optional optimizations:

1. Pools reachable from global variables need special handling for performance. The problem is that such a pool would be created in the function `main` , and would have to passed down through many layers of function calls to be available at each function that actually allocates or frees data in the pool. We use a simple solution, which we describe only briefly here for lack of space. We simply create a global variable to hold the pool descriptor for each heap node reachable from a global, and then use the appropriate global at every point where such a pool descriptor is needed.

2. We use a simple intraprocedural dataflow analysis to move `poolcreate` and `pooldestroy` calls later and earlier within the body of the function, so that the lifetime of objects in the pool is reduced. This optimization can also make it more likely that the next refinement can apply.

3. Our final optimization eliminates `poolfree` for a pool, P, if there are no allocations out of *any* pool (or `malloc` s) between the `poolfree` and the `pooldestroy` for pool P. Because there are no intervening allocations, the peak memory consumption of the program is not increased, even though the lifetime of the memory in pool P may have been increased. For example, in Figure 1(b), we can eliminate the call to `poolfree(&PD1, A)`, which also causes the compiler to eliminate the enclosing while loop. The same steps can be applied to `poolfree(&PD2, B)` as well.

5 Heuristic Choices in Segregating Data Structures

The algorithm of the previous section provides a general framework for segregating data structures in the heap. There are two aspects of the transformation where the algorithm makes specific choices, and several different options are available for each of those choices. First, the algorithm creates pools for all **H** DS nodes. Second, it creates creates distinct pools for each such DS node. Below, we motivate and discuss alternative heuristics that different clients might prefer for each of these two choices. Since heap objects are laid out separately and dynamically within each pool, these choices give the compiler some control over the internal layout of data structures.

5.1 Selecting Nodes to Assign to Pools

The core transformation can be modified to select a subset of candidate nodes to assign to pools. We simply change line (9) of Figure 4 so that, for DS nodes not assigned to a pool, no `poolcreate` or `pooldestroy` is created, and the pool descriptor is initialized to NULL. For a NULL pool, `poolalloc` and `poolfree` simply call `malloc` and `free` .

The choice of which candidate nodes to assign to pools depends on the client's needs. For the static heap safety enforcement technique described in [17], the previous choice (of allocating all nodes to pools) is necessary. If performance is the primary goal, however, the small overheads of pool creation and destruction are unnecessary for some pools. If a pool will hold only a single malloc object of any type (a

common case is a single array), pool allocation will not change the data layout. More generally, a pool only changes performance behavior if two or more objects in the pool can be visited by a single traversal.

To identify traversable objects using the DS graph, note that non-trivial traversals can only be made through array or pointer-based data structures, which will appear as cycles in the graph (except in degenerate cases). We therefore define a *Collection* to be either a node with the **A** set, or any non-trivial strongly connected component (SCC) in the DS Graph. A non-trivial SCC is one containing at least one cycle, including self-cycles. Now, 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. (Note that a single **A** node not in a cycle is not reachable from a collection, and so will not meet our criterion, which is what we want.)

Based on these principles and definitions, we propose two possible policies for choosing when to create a new pool for a canddiate **H** node in its parent function:

AllPools: In each function F, create a pool for every candidate **H** node for which F is the parent function.

TraversablePools: In each function F, create a pool for every candidate **H** node reachable from a collection.

5.2 Heuristics for Merging Pools

Our transformation so far creates a distinct static pool for every **H** node selected for assigning to pools. By merging pools together for distinct nodes, the compiler obtains partial control over the internal layout of a data structure. Given some DS subgraph that represents a high-level data structure of interest, the compiler can choose how to partition that subgraph into multiple pools. The objects within a particular pool can then be controlled in different ways via appropriate extensions to the pool allocation library. By default, objects might simply be interleaved in memory in allocation order. More sophisticated techniques might segregate objects into "bins" based on arbitrary compiler information (e.g., as used in [9]), simply by passing an extra bin parameter to the allocation routines.

The transformation algorithm is flexible enough to allow a common pool to be created for an arbitrary group of **H** nodes. We retain separate pool descriptor pointer variables for each **H** DS node, but simply change line 9 again so that `poolcreate` and `pooldestroy` are inserted for only one of the nodes, and the other pool descriptor pointers are initialized to point to the common descriptor.

We propose three potentially useful heuristics for choosing which **H** nodes should share a common pool:

OnePoolPerNode : Each candidate **H** nodes within a single function is assigned to a separate pool. This is particularly useful for clients that require type-homogeneous pools, including both the memory safety enforcement work [17]) and a "pointer compression" optimization mentioned in Section 9.

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. In particular, it is not uncommon that a program creates a complex, connected data structure consisting of multiple DS nodes and traverses it in the same order as it was created and putting all nodes in a single pool could allow such a traversal to be linear in memory (if objects are laid out by the library in allocation order).

The above heuristics are designed to work in the absence of profile information. In practice, grouping objects based on some knowledge of the patterns of traversal over data structures could significantly improve performance. Profile-driven heuristics could be used with the framework without any significant change to the core allocation algorithm itself, though doing that is outside the scope of this paper.

6 Uses of Pool Allocation

To illustrate how the layout informaton and control provided by automatic pool allocation could enable novel compiler transformations, and optimizations, we very briefly list a few examples of such techniques that fundamentally depend on pool allocation. The first of these is fairly mature [17], the second partially implemented, and the rest are speculative. Describing these techniques is outside the scope of this paper but are the subject of our future work.

Program safety without garbage collection: Our major application of pool allocation to date has been on enforcing program safety without garbage collection for type-safe C progams [17]. The key property this work exploits is that pools for type-safe data structures are type homogeneous. This allows us to enforce memory safety (but not type safety) without garbage collection, by ensuring that any dangling pointers into the pool will have the same target type as any new item allocated from the pool.

Transparent pointer compression: Allocating a data structure in a set of type homogeneous pools allows us to use *indexes into the pool* to identify objects, instead of full pointers. Since most data structures are likely to have far less than 2^{32} distinct nodes of any one type, these indexes could be much smaller (e.g., 16 or 32 bits) than a full pointer. This could reduce both the cache footprint and memory bandwidth of pointer-intensive data structures, especially on 64-bit machines. We have implemented a prototype of this transformation and are evaluating it currently.

Linear prefetching for linked data structures: As noted earlier, pool allocation converts some potentially unpredictable memory traversals into simple, linear traversals. In machines without stride prefetching hardware, the compiler could use a simple software prefetching algorithm similar to those used for dense arrays [10, 30], which are much more practical than algorithms for prefetching linked data structures (e.g., using jump pointers) [29].

Rewriting "complete" data structure traversals to operate *in pool order*: A much more sophisticated optimization could be possible for traversals that perform commutative operations [31] on *all* the nodes of a data structure (e.g., summing the values in a list, or finding the largest element in a tree). If we can identify such cases, we could transform the loop to simply iterate over the items in their order in the pool instead of in the original order. Reordering a traversal according to the layout of a complex data structure in memory is a non-trivial problem, and would be quite difficult without a combination of Automatic Pool Allocation (to expose the data structure layout) and DSA with some additional analysis (to decide when this is legal).

7 Results

We evaluated the direct performance impact of the Automatic Pool Allocation transformation for a number of heap-intensive and ordinary benchmark programs. We used two different combinations of heuristics described in Section 5. One is the core transformation where all **H** nodes are pool allocated in separate pools, i.e., **AllPools+OnePoolPerNode**. The second uses the combination **TraversablePools+OnePoolPerCollection**. We call these AllPools (AP) and CollectionPools (CP) respectively.

We used programs from the Olden, Ptrdist, and SPEC CINT 2000[3] benchmark suites, as well as analyzer from FreeBench 1.03 suite and llu-bench, an improved version of Olden-Health. (Other FreeBench programs showed no significant affects on performance and are omitted for lack of space.) We compiled

[3]Four SPEC CINT benchmarks were not working correctly in time for submission.

Program	LOC	Base runtime	PA AP runtime	PA CP runtime	Speedup ratio AP	Speedup ratio CP	Static Pools	Type Safe	Dyn. Pools	Num Args
bh	2091	12.484	12.394	12.280	1.007	1.017	2	1	2	0
bisort	348	13.113	12.849	12.926	1.021	1.014	1	1	1	1
em3d	682	6.299	6.289	6.334	1.002	0.994	8	7	8	45
health	508	8.676	7.168	7.545	1.210	1.150	2	2	2	4
mst	432	7.601	6.946	7.006	1.094	1.085	5	5	5	0
perimeter	484	1.862	1.588	1.601	1.173	1.163	1	1	1	1
power	622	5.680	5.773	5.731	0.984	0.991	4	4	4	5
treeadd	245	7.172	7.039	7.065	1.019	1.015	1	1	1	1
tsp	579	5.155	5.223	5.260	0.987	0.980	1	1	1	1
voronoi	1111	6.447	6.842	6.752	0.942	0.955	3	2	3	1
anagram	647	1.467	1.512	1.486	0.970	0.987	4	3	3	1
bc	7297	1.512	1.070	1.350	1.413	1.120	26	24	20	15
ft	1803	2.389	1.778	1.788	1.344	1.336	4	4	4	4
ks	782	2.133	2.152	2.149	0.991	0.993	3	3	3	0
yacr2	3982	3.280	3.157	3.321	1.039	0.988	26	26	26	0
164.gzip	8616	22.151	23.183	22.635	0.955	0.979	7	6	7	2
175.vpr	17729	9.230	9.732	9.459	0.948	0.976	153	139	44	33
181.mcf	2412	25.307	25.715	26.240	0.984	0.964	2	2	2	0
186.crafty	20650	20.326	21.156	20.979	0.961	0.969	5	5	4	0
197.parser	11391	5.551	5.717	5.728	0.971	0.969	1	0	1	0
197.parser(b)	11204	7.642	6.160	6.334	1.241	1.207	52	51	6675	86
255.vortex	67220	6.110	6.293	6.455	0.971	0.947	2	1	1	6
256.bzip2	4647	26.291	26.115	26.056	1.007	1.009	10	9	8	0
300.twolf	20459	9.344	8.986	9.636	1.040	0.970	111	106	231	6
analyzer	923	56.643	28.390	28.621	1.995	1.979	8	8	8	0
llu-bench	187	12.129	9.729	9.699	1.247	1.251	2	2	2	0

Table 1: Runtimes and statistics for Olden, Ptrdist, SPECINT2000, and other benchmarks

and optimized each program with LLVM, applied pool allocation at link-time (with one of the two heuristics), and used the LLVM interprocedural constant propagation and dead argument elimination passes after pool allocation to clean up unused pool arguments due to NULL pool descriptors. To get the "Base runtime", we used the exact same steps with the exception of the pool allocator, including the cleanup passes. We compiled the transformed LLVM code to C using the LLVM C backend and compiled the resultant C code with GCC 3.3, and ran all programs on an Intel 3.06GHz Xeon CPU.

Table 1 shows speedups and several other statistics. The fourth column group shows statistics about pools with the (AP) heuristic. This includes the static number of pools identified in the program, the number of pools that are type-homogenous, the dynamic number of pools created in a run of the program, and the number of arguments added by the pool allocator to the program.

The speedup results show that heap and data-structure intensive programs are sped up significantly by the pool allocation transformation, from 10-40% in several cases and up to 97% in `analyzer` from the FreeBench 1.03 suite), In contrast, some programs that have only a few pools are largely unaffected, and in particular are not hurt by the transformation. The large speedups come primarily from two sources: Improved cached behavior of data structure traversals (see below), and efficient memory allocation from type-homogenous pools. In particular, even though the Olden benchmarks have a small number of pools, almost all of their pools are type-homogenous.

Many of the SPEC benchmarks are heavily optimized to allocate and deallocate from the heap as infrequently as possible. `181.mcf`, for example, allocates large arrays of nodes whose elements it manages by itself. Pool allocation has no benefit (but also does not hurt) this program. Several other benchmarks including `197.parser`, `255.vortex`, and `175.vpr` use custom memory allocators, which obscure the memory allocation behavior of the program. In order to evaluate the impact of this problem, we changed 197.parser to call malloc/free directly, yielding the `197.parser(b)` benchmark. Because it calls malloc & free, DSA is able to identify logical datastructures, and the pool allocator is then able to segregate them. Notice that the pool allocator is almost able to match the performance of the hand tuned allocator, while segregating data structures and exposing type information about them to the compiler.

Figure 6 shows the change in cache misses for each program in the Table 1 that shows a speedup of more than 3%. From the figure, we can see that the program with the largest speedup (`analyzer`) has a very large reduction of both L1 and L2 cache misses. Other benchmarks (such as `perimeter` and `ft`) speed up primarily due to better utilization of the L1 cache even in the case of `ft` at the expense of a

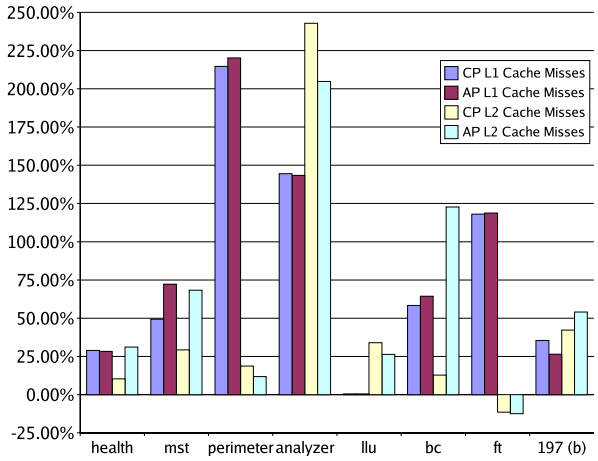

Figure 6: L1/L2 Cache Miss Reduction (percent)

few more L2 cache misses. We believe (but have yet to prove) that the pool allocation transformation is turning pointer chasing data structure traversals into regular-stride memory accesses that the hardware prefetcher on the Intel Pentium 4 Xeon processor can make use of. In addition, segregating memory objects from disjoint data structures allows for a natural increase in effective cache density.

This chart also shows that the CP mechanism is failing to pool allocate important data structures in some benchmarks, such as the `bc` benchmark from the Ptrdist suite. With further refinement and experimentation we hope to tune the heuristic to be able to pool allocate all of the important data structures while eliminating the obviously uninteresting ones (for example, pools where only a single memory object is allocated).

8 Related Work

As noted in the Introduction, there are two major differences between the work region inference for ML [37, 36, 21] and our work. More specifically: (1) Although they can allocate distinct aggregate values (e.g., a list) to separate regions, they do not identify and control the internal layout of complex data structures (e.g., a list of trees constructed as a single value will be assigned a single region). Our algorithm tracks objects and the pointers between them separately to enable internal segregation. (2) Their algorithm does not perform any (interprocedural) points-to analysis and therefore deals with destructive heap updates (storing values in reference types) in a limited manner. In particular, when storing a value into a non-local reference, the lifetime of the reference is not directly tracked, and in fact can cause the memory for that value to be leaked [36]. In contrast, our algorithm is driven by an interprocedural points-to representation and allows explicit (though conservative) tracking of lifetimes of objects, which is essential for languages with extensive use of destructive heap updates.

Aiken et al. [1] show how to relax another assumption in the Tofte and Birkedal transformation, to allow regions to have arbitrary non-nested lifetimes. Our placement optimization for regions is conceptually similar, but we do not consider placements across procedure boundaries (though our algorithm could be extended to do so).

Several recent languages have included mechanisms for manual region-based memory as an alternative to garbage collection, e.g., Real-time Java [7], Cyclone [24] and others [19, 15, 8]. In these languages, the heap is partitioned into separate regions and deallocation is only permitted on an entire region at once. Compared with our work, these language-based techniques are much easier to implement, but require significant manual effort to use. More importantly for our work, although these 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. Similarly, runtime libraries for region-based memory management [35, 4] also require manual effort and do not provide the compiler any information to support further analyses or transformations.

There is a wide range of work on techniques for stack allocation of heap objects (e.g., see the recent paper by [6] and the references therein). Unlike our work and the region inference work above, stack allocation does not modify any properties of the remaining heap data. It does not attempt to analyze or control the layout of logical data structures for either class of data. Similarly, techniques for static garbage collection [33, 25] must perform similar analyses to identify object lifetimes and decide statically when data can be freed, but they do not control data structure layout.

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` is a `malloc` replacement that accepts hints to allocate one object near another object, but this only provides local information for an object pair and not any global information about entire data structures.

Hirzel et al. [23] describe a technique to improve the effectiveness of GC 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 is likely to make it them much more coarse grain than nodes in DS graphs.

Several proposed techniques attempt to improve storage allocation or garbage collection performance by relating objects based on their predicted lifetimes [22, 16, 3, 14, 13, 32]. 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 [20, 9, 32, 11]. These placement decisions are orthogonal to the choices made by Automatic Pool Allocation, and could therefore be combined with our transformation. In fact, we consider this an important direction for future work.

Finally, in a preliminary workshop paper, we proposed the basic idea of Automatic Pool Allocation, but with an impractical algorithm and without any evaluation. That work did not consider how to handle the key difficult cases for this transformation, namely, function pointers, type-unsafe data structures, and globally live pools. It also relied on an exponential version of the Data Structure Analysis algorithm, but making DSA efficient [26] introduced key complications for pool allocation, such as collapsed nodes. Finally, the previous paper was too early to present any performance evaluation. We believe the transformation described here greatly improves over the previous one, addressing all the challenges above and making it both practical and efficient. We have also refined several algorithmic choices through our experience using it in our safety checking work [17].

9 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 programs. Our results showed that for many programs, the transformation achieves the first of the two goals stated in the Introduction, namely, that it can improves their performance, sometimes very substantially.

We have two goals in our ongoing work. First, we wish to compare the effectiveness of automatic pool

allocation with alternative pointer analysis algorithms (e.g., DSA vs. Anderson's algorithm). The second and broader aim of our work is to implement and evaluate some of the novel optimization techniques described in Section 6. We believe that the combination of Data Structure Analysis and Automatic Pool Allocation together provide a powerful foundation for analyzing and transforming programs *in terms of how they create and use entire logical data structures*, rather than in terms of individual memory references or data elements.

References

[1] A. Aiken, M. Fähndrich, and R. Levien. Better static memory management: Improving region-based analysis of higher-order languages. In *Proc. ACM SIGPLAN '95 Conf. on Programming Language Design and Implementation*, pages 174–185, June 1995.

[2] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994.

[3] 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.

[4] E. D. Berger, B. G. Zorn, and K. S. McKinley. Reconsidering custom memory allocation. In *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA) 2002*, Seattle, Washington, Nov. 2002.

[5] L. Birkedal, M. Tofte, and M. Vejlstrup. From region inference to von neumann machines via region representation inference. In *Proc. 23rd ACM Symp. on Principles of Prog. Languages*, pages 171–183, 1996.

[6] B. Blanchet. Escape analysis for java(tm): Theory and practice. *ACM Transactions on Programming Languages and Systems*, 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 *SIGPLAN Conference on Programming Language Design and Implementation*, 2003.

[9] B. Calder, K. Chandra, S. John, and T. Austin. Cache-conscious data placement. In *Proc. 8^{th} Int'l Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, pages 139–149, San Jose, USA, 1998.

[10] D. Callahan, K. Kennedy, and A. Porterfield. Software prefetching. In *Proc. 4^{th} Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 40–52, Santa Clara, USA, Apr. 1991.

[11] T. M. Chilimbi, B. Davidson, and J. R. Larus. Cache-conscious structure definition. In *ACM Symp. on Prog. Lang. Design and Implemenation*, Atlanta, GA, May 1999.

[12] T. M. Chilimbi, M. D. Hill, and J. R. Larus. Cache-conscious structure layout. In *PLDI*, Atlanta, GA, May 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] R. Courts. Improving locality of reference in a garbage-collecting memory management system. *Communications of the ACM*, 31(9):1128–1138, 1988.

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

[16] 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 Symposium on Principles of Prog. Lang.*, pages 261–269, 1990.

[17] D. Dhurjati, S. Kowshik, V. Adve, and C. Lattner. Memory safety without runtime checks or garbage collection. In *Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, San Diego, Jun 2003.

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

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

[20] D. Grunwald and B. G. Zorn. Customalloc: Efficient synthesized memory allocators. *Software - Practice and Experience*, 23(8):851–869, 1993.

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

[22] D. R. Hanson. Fast allocation and deallcoation of memory based on object lifetimes. *Software Practice and Experience*, 20(1):5–12, Jan 1990.

[23] M. Hirzel, A. Diwan, and M. Hertz. Connectivity-based garbage collection. In *Proceedings of the 18th ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 359–373, 2003.

[24] 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.

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

[26] C. Lattner and V. Adve. Data Structure Analysis: A Fast and Scalable Context-Sensitive Heap Analysis. Tech. Report UIUCDCS-R-2003-2340, Computer Science Dept., Univ. of Illinois at Urbana-Champaign, Apr 2003.

[27] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. In *Proc. 2004 Int'l Symposium on Code Generation and Optimization*, San Jose, USA, Mar 2004.

[28] D. Liang and M. J. Harrold. Efficient computation of parameterized pointer information for interprocedural analysis. In *Static Analysis Symposium*, 2001.

[29] C. Luk and T. Mowry. Compiler-based Prefetching for Recursive Data Structures. In *ASPLOS*, Boston, Oct. 1996.

[30] T. Mowry, M. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Proc. 5^{th} Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)*, pages 62–73, Boston, USA, Oct. 1992.

[31] M. C. Rinard and P. C. Diniz. Commutativity analysis: a new analysis technique for parallelizing compilers. *Trans. Prog. Lang. and Systems*, 19(6):942–991, 1997.

[32] M. L. Seidl and B. G. Zorn. Segregating heap objects by reference behavior and lifetime. In *Proc. 8^{th} Int'l Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, pages 12–23, San Jose, USA, 1998.

[33] R. Shaham, E. Yahav, E. K. Kolodner, and M. Sagiv. Establishing local temporal heap safety properties with applications to compile-time memory management. In *Int'l Static Analysis Symposium*, San Diego, USA, June 2003.

[34] B. Steensgaard. Points-to analysis in almost linear time. In *Symposium on Principles of Programming Languages*, pages 32–41, Jan 1996.

[35] A. Stepanov and M. Lee. The Standard Template Library. Technical Report X3J16/94-0095, WG21/N0482, 1994.

[36] M. Tofte and L. Birkedal. A region inference algorithm. *Trans. Prog. Lang. and Systems*, 20(4):724–768, 1998.

[37] M. Tofte and J.-P. Talpin. Implementation of the typed call-by-value λ-calculus using a stack of regions. In *Proc. 21^{st} Annual ACM Symposium on Principles of Programming Languages*, pages 188–201, 1994.