Automatic pool allocation

Introduction

Chenhao Li, Denghang Hu, Lv Feng

University of Chinese Academy of Sciences

July 12, 2018

Outline

What is APA

The full name of *APA* is *automatic pool allocation*.

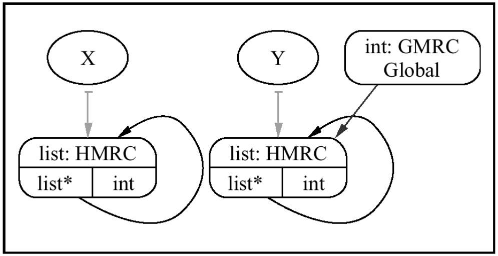
A transformation framework that segregates distinct instances of heap-based data structures into seperate memory pools and allows heuristics to be used to partially control the internal layout of those data structures.

For example, each distinct instance of a list, tree, or graph identified by the compiler would be allocated to a separate pool.

What is APA

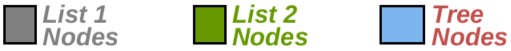
- Segregate memory according to points-to graph
- Use context-sensitive analysis to distinguish between RDS instances passed to common routines

Points-to graph (two disjoint linked lists)

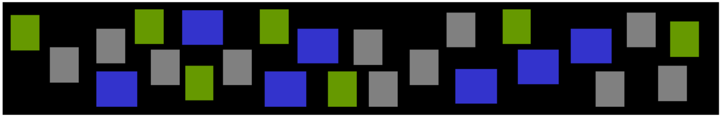


Problem

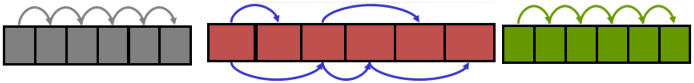
Apa is a backend optimize method in LLVM.



What the compiler sees



What we want the program to create and the compiler to see



Problem

- **Memory system performance is important!**
  - Fast CPU, slow memory, not enough cache
- **"Data structures" are bad for compilers**
  - Traditional scalar optimizations are not enough
  - Memory traffic is main bottleneck for many apps
- **Fine grain approaches have limited gains:**
  - Prefetching recursive structures is hard
  - Transforming individual nodes give limited gains

Compute Disjoint Data Structure Graphs

- A significant part in poolalloc is computing disjoint data structure graphs.
- We use an algorithm called Data Structure Analysis (DSA) to compute these disjoint data structure graphs.
- **Properties of DSA:**
  - context-sensitive(malloc nodes of two distinct lists in a same point)
  - unification-based(simplification, may-point-to)
  - field-sensitive(avoid merging the target of unrealted pointer field)

DSA

In DSA, the key analysis informations we use is as follows:

- SSA form: assume a low-level code representation with an infinite set of virtual registers, and a load-store architecture
- Identification of memory objects: heap objects allocated by malloc, stack objects allocated by alloca, global variables, and functions;
- Type information: we assume that all SSA variables and memory objects have an associated type.
- Safety information: our analysis requires that there is some way to distinguish type-safe and type-unsafe usage of data values.

Disjoint Data Structure Graph

Elements:

- Node
  - each node represents a typed SSA register or a memory object allocated by the program, or multiple objects of the same type.
  - A node is represented by a node type(new node/alloca node/global node/function node/call node/shadow node/cast node/scalar node)
- Edge
  - Each edge in the graph connects a pointer field of one node (the source field) to a field of another node (the target field).
  - A pointer field may have edges to multiple targets, i.e., edges represent "may-point-to" information.

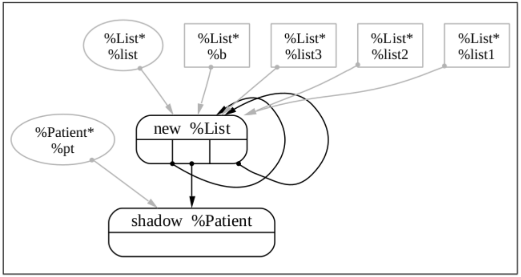
Example

```
/* C Source Code */
struct Patient { ... };
typedef struct List {
  struct List    *forward;
  struct Patient *patient;
  struct List    *back;
} List;

void addList(List *list, struct Patient *pt) {
  List *b = NULL;
  while (list != NULL) {
    b = list;
    list = list->forward;
  }
  list = (List *)malloc(sizeof(List));
  list->patient = pt;
  list->forward = NULL;
  list->back    = b;
  b->forward    = list;
}
```

```
;; LLVM assembly code
%Patient = type { ... }
%List    = type { %List*, %Patient*, %List* }

void %addList(%List* %list, %Patient* %pt) {
bb0:
  %cond1 = seteq %List* %list, null
  br bool %cond1, label %bb3, label %bb2
bb2:
  %list1 = phi   %List* [%list2,%bb2], [%list,%bb
  %list2 = load  %List* %list1, uint 0
  %cond2 = setne %List* %list2, null
  br bool %cond2, label %bb2, label %bb3
bb3:
  %b     = phi   %List* [%list1,%bb2], [null,%bb0
  %list3 = malloc %List
  store %Patient* %pt,    %List* %list3, uint 1
  store %List*    null,   %List* %list3, uint 0
  store %List*    %b,     %List* %list3, uint 2
  store %List*    %list3, %List* %b,     uint 0
  ret void
}
```

Example

Data Structure Graph for *addList*:



In our graphs, the dark rounded objects represent actual memory objects that exist in the program, whereas the lighter objects represent scalar values in the function.

Intraprocedural Analysis Algorithm

- The intraprocedural graph computation phase is a flow-insensitive analysis that builds a data structure graph without requiring the code for other functions to be available.
- The graph construction algorithm is composed of three distinct phases:
  - the node discovery phase
  - the worklist processing phase
  - and the graph simplification phase

Node Discovery Phase

- Performs a single **pass** over the function being processed, creating the nodes that make up the graph.
- The worklist processing phase can only add new shadow nodes and edges to the graph, so all nodes of other types come from the node discovery phase.

Worklist processing

- The worklist contains all of the instructions in the function that use the SSA values corresponding to the nodes.
- Processing:

Algorithm 1 ProcessWorkList

```
1: while $WL \neq \varnothing$ do
2:     instruction $inst = WL.head$
3:     $WL$.remove($inst$)
4:     $process\ instruction(inst)$
5: end while
```

- Worklist processing phase will create shodow nodes and add edges between nodes according to points-to relations.

Graph Simplification

- Merge indistinguishable nodes(edge points change). Two nodes are considered indistinguishable if they are of the same LLVM type and if there is a field in the data structure graph that points to both nodes..
- Pool allocation actually benefits from graphs that are merged as much as possible, as long as two disjoint structures are not unneccesarily merged together.

Interprocedural Closure Algorithm

- Local analysis graph is of limited usefulness:
  - Intraprocedural analysis only uses code available in current function.
  - Most interesting data structures are passed to funcion to construct or manipulate them.

  Therefore, we can not know those datastructures' type, transformation becomes impossible.
- Interprocedural Closure:
  - Inline information of called funcions to the caller function's graphs.

Interprocedural Closure Algorithm

Special Cases:

- Indirect calls: repeatedly inlining the called function graphs for each function called by a particular call node, that is recursively inlining the indirect call.
- Recursive function: The result of inlining a function call is memoized in the InlinedFnsSet to avoid infinite recursion when inlining recursive functions.
- Mutually recursive function: calculate the interprocedural closure graphs in a postorder traversal over the call graph.

Runtime Support

- A simple pool allocation runtime library with four external functions:
  - poolinit
  - pooldestroy
  - poolalloc
  - poolfree
- Our pool allocator assumes that a memory pool consists of uniformly sized objects, but can allocate multiple consecutive objects if needed (for arrays).
- When pool allocating a complex data structure, each data structure node in the graph is allocated from a different pool in memory.

Identifying candidate data structures

- In order to pool allocate a data structure, we must detect the bounds on the lifetime of the data structure (to allocate and delete the pools themselves), and determine whether it is safe to pool allocate the data structure.
- Using the data structure graph, we detect data structures whose lifetimes are bound by a function lifetime, allowing us to allocate the pool on entry to the function, and deallocate it on exit from the function.
- Each function's graph only contains the data structures that are acessable by that function, so we identify these candidates by scanning the functions in the program

Candidate identification algorithm

Algorithm 2 PoolAllocateProgram

```
1: **for** each function $Fn \in Prog$ **do**
2:     **for** each disjointdatastructure $DS \in DSGraph(F_n)$ **do**
3:         **if** CallNodes($DS$) $\cup$ CastNodes($DS$) $= \varnothing$ **then**
4:             **if** $\neg$ escapes(DS) **then**
5:                 PoolAllocate($Fn, DS$)
6:             **end if**
7:         **end if**
8:     **end for**
9: **end for**
```

Escape

having globals point to the structure, or it is returned from the current function.

Transforming function bodies

Algorithm 3 PoolAllocate

```
Require: funcion $RootFn$, datastructure $DS$
 1: $Worklist = \{RootFn\}$
 2: for each function $Fn \in Worklist$ do
 3:     for each instruction $I \in$ Instructions($Fn$) do
 4:         if UsesDataStructure($I, DS$)) then
 5:             if IsMallocOrFree(I) then
 6:                 ConvertToPoolFunction($I, DS$)
 7:             else if IsCall(I) then
 8:                 AddPoolArguments($I, DS$)
 9:                 $Worklist = Worklist \cup$CalledFunction($I$)
10:             end if
11:         end if
12:     end for
13: end for
```

Transforming funcion bodies

- *RootFn*'s lifetime bounds the lifetime of *DS*
- The transformation loops over a worklist of functions to process, transforming each function until the worklist is empty.
- **malloc** and **free** operations referring to the pool allocated data structure are changed into calls to the **poolalloc** and **poolfree** library functions.

Experiment

- We use *LLVM* 3.3 and *poolalloc* *r*192788 to experiment.
- Little resources about how to use *poolalloc* on the Internet.
- Finally, we solved all the problem in using *poolalloc* thanks to lots of useful help from Prof.Cui and assistant Zhao.

Using DSA

DSA roughly works in three distinct phases:

- Local: nodes discovery and create nodes. local analysis is run on each function in the program, creating separate graphs for each.
- Bottom-Up: run after the local phase. Iterates over the callgraph, callees before callers, and inlines the callee's graph into the caller
- Top-Down: iterates over the callgraph again, this time callers before callees, and merges nodes in callees when necessary.

Using DSA

- Using *opt* to load *LLVMDataStructure.so* and *poolalloc* libraries to transform on LLVM bytecode.
- Generating bytecode:

```
$ clang −emit−llvm −c test.c
```

- DSA analyze(take *Bottom* − *Up* analysis as example):

```
$ opt −load  /path/to/LLVMDataStructure.so −load
/path/to/poolalloc.so −analyze  −dsa−bu test.o
```

- After last step, it will gennerate .dot files of all the disjoint datastructures.
- Visualization: using *dot* program to generate pdf or png file from .dot file.

Using poolalloc

- Optimize:

```
$ opt -load /path/to/LLVMDataStructure.so -load
/path/to/poolalloc.so -poolalloc test.o > test.o1
```

- Generate visual bytecode:

```
$ llvm-dis test.o/test.o1
```

- We can check the differ between not optimized bytecode and optimized bytecode:

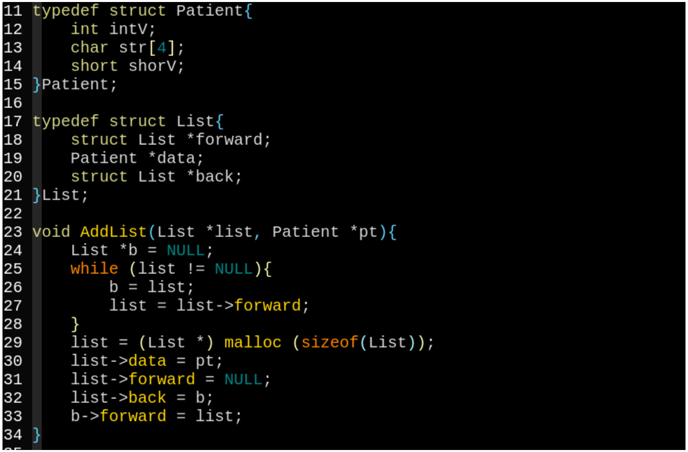
```
$ diff test.o.ll test.o1.ll
```

- Generate executable file:

```
$ llc test.o1.ll
$ gcc /path/to/libpoolalloc_rt.a test.o1.s
```

Example

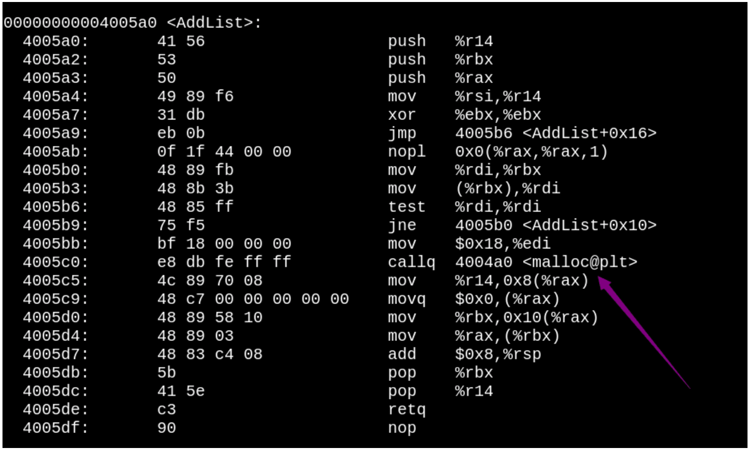
A simple function *AddList*:



```
typedef struct Patient{
    int intV;
    char str[4];
    short shorV;
}Patient;

typedef struct List{
    struct List *forward;
    Patient *data;
    struct List *back;
}List;

void AddList(List *list, Patient *pt){
    List *b = NULL;
    while (list != NULL){
        b = list;
        list = list->forward;
    }
    list = (List *) malloc (sizeof(List));
    list->data = pt;
    list->forward = NULL;
    list->back = b;
    b->forward = list;
}
```

Before poolalloc

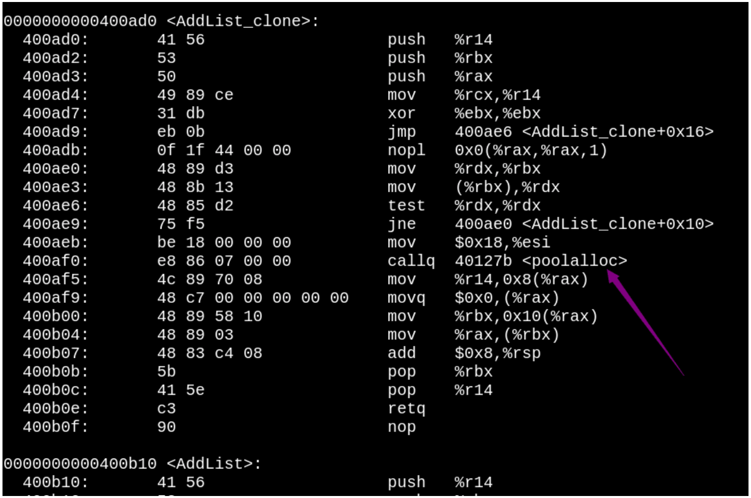
Objdump result of binary file compiled with *clang -O3*:



```
00000000004005a0 <AddList>:
  4005a0:       41 56                   push   %r14
  4005a2:       53                      push   %rbx
  4005a3:       50                      push   %rax
  4005a4:       49 89 f6                mov    %rsi,%r14
  4005a7:       31 db                   xor    %ebx,%ebx
  4005a9:       eb 0b                   jmp    4005b6 <AddList+0x16>
  4005ab:       0f 1f 44 00 00          nopl   0x0(%rax,%rax,1)
  4005b0:       48 89 fb                mov    %rdi,%rbx
  4005b3:       48 8b 3b                mov    (%rbx),%rdi
  4005b6:       48 85 ff                test   %rdi,%rdi
  4005b9:       75 f5                   jne    4005b0 <AddList+0x10>
  4005bb:       bf 18 00 00 00          mov    $0x18,%edi
  4005c0:       e8 db fe ff ff          callq  4004a0 <malloc@plt>
  4005c5:       4c 89 70 08             mov    %r14,0x8(%rax)
  4005c9:       48 c7 00 00 00 00 00    movq   $0x0,(%rax)
  4005d0:       48 89 58 10             mov    %rbx,0x10(%rax)
  4005d4:       48 89 03                mov    %rax,(%rbx)
  4005d7:       48 83 c4 08             add    $0x8,%rsp
  4005db:       5b                      pop    %rbx
  4005dc:       41 5e                   pop    %r14
  4005de:       c3                      retq
  4005df:       90                      nop
```

After poolalloc

Objdump result after poolalloc:



```
0000000000400ad0 <AddList_clone>:
  400ad0:       41 56                   push   %r14
  400ad2:       53                      push   %rbx
  400ad3:       50                      push   %rax
  400ad4:       49 89 ce                mov    %rcx,%r14
  400ad7:       31 db                   xor    %ebx,%ebx
  400ad9:       eb 0b                   jmp    400ae6 <AddList_clone+0x16>
  400adb:       0f 1f 44 00 00          nopl   0x0(%rax,%rax,1)
  400ae0:       48 89 d3                mov    %rdx,%rbx
  400ae3:       48 8b 13                mov    (%rbx),%rdx
  400ae6:       48 85 d2                test   %rdx,%rdx
  400ae9:       75 f5                   jne    400ae0 <AddList_clone+0x10>
  400aeb:       be 18 00 00 00          mov    $0x18,%esi
  400af0:       e8 86 07 00 00          callq  40127b <poolalloc>
  400af5:       4c 89 70 08             mov    %r14,0x8(%rax)
  400af9:       48 c7 00 00 00 00 00    movq   $0x0,(%rax)
  400b00:       48 89 58 10             mov    %rbx,0x10(%rax)
  400b04:       48 89 03                mov    %rax,(%rbx)
  400b07:       48 83 c4 08             add    $0x8,%rsp
  400b0b:       5b                      pop    %rbx
  400b0c:       41 5e                   pop    %r14
  400b0e:       c3                      retq
  400b0f:       90                      nop

0000000000400b10 <AddList>:
  400b10:       41 56                   push   %r14
```

LLVM Pass

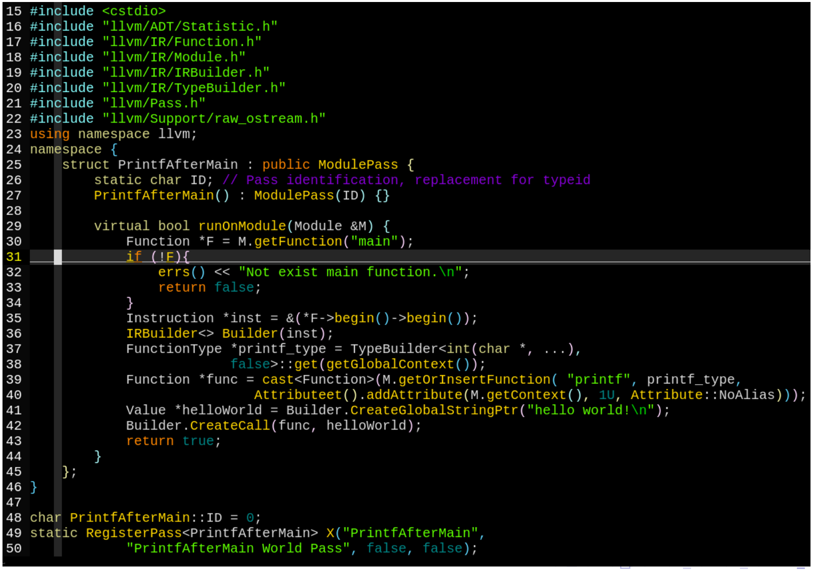
What is LLVM Pass

- The LLVM Pass Framework is an important part of the LLVM system. Passes perform the transformations and optimizations that make up the compiler, they build the analysis results that are used by these transformations.
- All LLVM passes are subclasses of the *Pass* class, which implement functionality by overriding virtual methods inherited from Pass.
- Depending on how your pass works, you should inherit from:
    - *ModulePass*
    - *FunctionPass*
    - *CallGraphSCCPass*
    - *LoopPass*
    - *RegionPass*
    - *BasicBlockPass*

LLVM Pass

- Passes can be dynamically loaded by the opt tool via its -load option.
- Passes are registered with the *RegisterPass* template. The template parameter is the name of the pass that is to be used on the command line to specify that the pass should be added to a program.

Example: a simple Pass inherit from the *ModulePass* which just add a *printf*(" *hello, world.*") at the beginning of *main* function if it exists:



```
#include <cstdio>
#include "llvm/ADT/Statistic.h"
#include "llvm/IR/Function.h"
#include "llvm/IR/Module.h"
#include "llvm/IR/IRBuilder.h"
#include "llvm/IR/TypeBuilder.h"
#include "llvm/Pass.h"
#include "llvm/Support/raw_ostream.h"
using namespace llvm;
namespace {
    struct PrintfAfterMain : public ModulePass {
        static char ID; // Pass identification, replacement for typeid
        PrintfAfterMain() : ModulePass(ID) {}

        virtual bool runOnModule(Module &M) {
            Function *F = M.getFunction("main");
            if (!F){
                errs() << "Not exist main function.\n";
                return false;
            }
            Instruction *inst = &(*F->begin()->begin());
            IRBuilder<> Builder(inst);
            FunctionType *printf_type = TypeBuilder<int(char *, ...),
                         false>::get(getGlobalContext());
            Function *func = cast<Function>(M.getOrInsertFunction( "printf", printf_type,
                            Attributeet().addAttribute(M.getContext(), 1U, Attribute::NoAlias)));
            Value *helloWorld = Builder.CreateGlobalStringPtr("hello world!\n");
            Builder.CreateCall(func, helloWorld);
            return true;
        }
    };
}

char PrintfAfterMain::ID = 0;
static RegisterPass<PrintfAfterMain> X("PrintfAfterMain",
            "PrintfAfterMain World Pass", false, false);
```

PoolAllocate

- file: poolalloc/lib/PoolAllocate/PoolAllocate.cpp
- Class PoolAllocate: public PoolAllocateGroup{//...}
- Class PoolAllocateGroup: public ModulePass{//...}
- Main method: runOnModule(Module & M){//...}, it corresponds to the *algorithm* above(Candidate identification and Transforming function)

bool PoolAllocate: :runOnModule(Module & M)

- $if(M.begin() == M.end())\ return\ false$; //Module maintains a functions list, Module is empty, so nothing need to do.
- $Graphs = \&getAnalysis < \cdots > ()$; //Accoring to code type, obtain corresponding DSA.
- $AddPoolProtypes(\&M)$ // add the pool* prototype to the Module, later we will replace all malloc/free with pool*.
- $GlobalPoolCtor = createGlobalPoolCtor(M); SetupGlobalPolls(M)$; // create a global pool and poolalloc all the global DSNodes(reachable from global)

bool PoolAllocate: :runOnModule(Module & M)

continue. . .

- *FindPoolArgs*(*M*); // Find the DSNodes for each function that will require pool descriptor arguments to be passed into the function.
- Transform functions: Not simply transforming all funcions need to poolalloc.
- In order to avoid iterator invalidation errors(random memory errors):
  - Clone all functions which need pool descriptor arguments(Add arguments.)
  - Transform all cloned functions or origin functions if the origin has no clone.
  - Replace any remaining uses of original functions with the transformed function.

References

- Chris Lattner and Vikram Adve. Automatic Pool Allocation for Disjoint Data Structures
- Chris Lattner and Vikram Adve. Automatic Pool Allocation: Improving Performance by Controlling Data Structure Layout in the Heap
- Writing an LLVM Pass
- *dsa-manual*