Backwards-Compatible Array Bounds Checking for C with Very Low Overhead

Dinakar Dhurjati  Vikram Adve
C bounds checking

- fat pointers
- not compatible for unchecked code
- separate metadata
- pointer-to-metadata map
- careful engineering allows compatibility
Automatic Pool Allocation

• merge all target objects of one pointer to a pool

• „pools will be type homogeneous with a known type“

• pools convey type information for pointers
Automatic Pool Allocation

Chris Lattner and Vikram Adve

Presented by William Lovas
Motivation

- Data locality is important!
- Compilers are good with arrays…
- … but bad with pointer-based data structures
Motivation

- Existing techniques focus on individual references or data elements

- Big idea: analyze how programs use *entire data structures*!
• Allocate disjoint data structures in disjoint portions of the heap (pools)

• … automatically, via static program transformation!
• Transform:
• Into:
Approach

- Create a *data structure graph* for each function $F$
  - A “points-to” graph with some extra info
- DS graph records, for each object:
  - Type of the object
  - Whether it’s heap_allocated
  - Whether it escapes $F$
Approach

- Use DS graph to assign a pool to each object
- Use assignment to rewrite program:
  - Calls to `malloc/free` become calls to `pool_alloc/pool_free`
  - Creates local pools for non-escaping objects
  - Adds pool arguments for escaping objects
Example [Lattner]

```c
list *makeList(int Num) {
    list *New = malloc(sizeof(list));
    New->Next = Num ? makeList(Num-1) : 0;
    New->Data = Num; return New;
}

void twoLists() {
    list *X = makeList(10);
    list *Y = makeList(100);
    GL = Y;
    processList(X);
    processList(Y);
    freeList(X);
    freeList(Y);
}
```
Example [Lattner]

list *makeList(int Num, Pool *P) {
    list *New = pool_alloc(P, sizeof(list));
    New->Next = Num ? makeList(Num-1, P) : 0;
    New->Data = Num; return New;
}

void twoLists() {

    list *X = makeList(10);
    list *Y = makeList(100);
    GL = Y;
    processList(X);
    processList(Y);
    freeList(X);
    freeList(Y);
}
Example [Lattner]

list *makeList(int Num, Pool *P) {
    list *New = pool_alloc(P, sizeof(list));
    New->Next = Num ? makeList(Num-1, P) : 0;
    New->Data = Num; return New;
}

void twoLists( ) {
    Pool P1;
    pool_init(&P1);
    list *X = makeList(10, &P1);
    list *Y = makeList(100);
    GL = Y;
    processList(X);
    processList(Y);
    freeList(X, &P1);
    freeList(Y);
    pool_destroy(&P1);
}
Example [Lattner]

list *makeList(int Num, Pool *P) {
    list *New = pool_alloc(P, sizeof(list));
    New->Next = Num ? makeList(Num-1, P) : 0;
    New->Data = Num; return New;
}

void twoLists( Pool *P2 ) {
    Pool P1;
    pool_init(&P1);
    list *X = makeList(10, &P1);
    list *Y = makeList(100, P2);
    GL = Y;
    processList(X);
    processList(Y);
    freeList(X, &P1);
    freeList(Y, P2);
    pool_destroy(&P1);
}
• Function pointers
  ▪ Two functions with *different properties* might be called (indirectly) at the *same site*

• Solution:
  ▪ Partition functions into equivalence classes
  ▪ Merge DS graphs
Difficulties

- Global pools
  - Pool arguments for heap-allocated globals must be added to every function that touches the globals
  - Can be thousands of arguments in practice

- Solution:
  - Use global variables for global pools
  - Pool arguments grow with original arguments
Results

• Small additional compile time
  ▪ <= 1.25 seconds in all experiments
  ▪ <= 3% of total compile time

• Low overhead
  ▪ <= 5% in most experiments

• Improved performance
  ▪ 5% to 20% in most experiments
  ▪ 2x and 10x in a few examples
• Limited discussion of corner cases

• Automatic pool allocation could decrease performance
  ▪ Decrease locality for certain access patterns
  ▪ Small pools on nearly-empty pages
  ▪ Some techniques help address these issues
Conclusions

- Simple yet sophisticated data structure analysis, for data locality
- Experimentally validated
- Not obviously universally applicable
Engineering

• use type-information provided by pooling to speed up pointer metadata search
• heavier verification for pointer arithmetics
• lightweight verification for pointer use
• track out-of-bounds pointers
Questions

• What errors cannot be detected?

• How „safe“ are we compared to Java / OCaml / …?

• Are the C-library wrappers for API-checking cheating?

• Usability of the approach?