PROMOTING HEAP ALLOCATIONS TO THE STACK IN LLVM

Daniel Anderson and Jatin Arora
Carnegie Mellon University

Overview

- Heap allocations can become a performance bottleneck in many programs.
- In some cases, users may use heap-allocated memory, perhaps unknowingly through the use of a library data structure, in a situation where the lifetime is obvious in advance.
- Moving suitable heap allocations to the stack can provide a speedup and enable further optimizations

Why the stack?

Allocating on the stack is more efficient when possible

- Fewer instructions: Moving the stack pointer is a single instruction, rather than the many required to perform `malloc` and `free`
- Less memory fragmentation: Memory allocated by `malloc` calls is reclaimed manually, if at all. This creates holes in memory.
- Better locality: Stack allocations are contiguous and experience better spatial locality. This reduces the cost of accessing the allocations

Removing mallocs enables further optimization

- Stack-allocated objects are more amenable to optimisations than heap allocated objects
- Analyses on objects that are stored in the heap are difficult as they generally require concrete heap models.
- Heap models are either very expensive or very abstract

Capture tracking

- An object on the heap can promoted to the stack if it does not outlive the scope of it's creation. Capture tracking is used for conservatively estimating the lifetime of objects
- **Capture tracking** determines for a given pointer, whether a copy of it is made that outlives the function that it was defined in.
- If the pointer is not captured, the object can not be accessed outside the scope of the function. Therefore, it is a candidate for stack promotion
- If a pointer is captured, its precise lifetime is hard or impossible to determine, so we conservatively assume that it is accessible outside the scope

```c
// A non-capturing function
int f(int* p) {
  return 2 + *p;
}
```

```c
// A capturing function
int f(int* p) {
  global_list.push_back(p);
  return *p - 3;
}
```

Fig. 1: A function that captures and one that does not

Cheeky mallocs

- Given a candidate for stack promotion, we need to determine whether it is safe to promote to the stack
- Main problem: Need to ensure that the memory is only deallocated in the scope in which it was allocated, so that we do not accidentally try to deallocate the stack memory!
- We say that a malloc call is **cheeky** if it is possible to reach an exit of the enclosing function without passing through a corresponding free
- If a malloc is not cheeky, we prove that we can safely convert it to a stack allocation and delete the corresponding frees

```c
// A non-cheeky malloc
int f() {
  int* p = (int*)malloc(16*4);
  p[0] = 1;
  for (int i = 1; i < 16; i++) {
    p[i] = p[i-1] + i;
  }
  int answer = p[15];
  free(p);
  return answer;
}
```

```c
// A cheeky malloc
int f() {
  int* p = (int*)malloc(16*4);
  // This function MIGHT free p
  int answer = compute(p);
  if (answer < 0) {
    free(p);
  }
  return answer;
}
```

Fig. 2: An example of a non-cheeky malloc and a cheeky malloc

Algorithm for finding cheeky mallocs

Algorithm 1 Finding Cheeky Mallocs

```
1:  procedure IsCheeky(m : Instruction)
2:    local visited[b] ← false for all basic blocks b
3:    procedure dfs(b : BasicBlock)
4:      visited[b] ← true
5:      for each Instruction I in b do
6:        if I = free(m) then
7:          return false
8:        else if I is a return instruction then
9:          return true
10:     for each BasicBlock s in Successors(b) do
11:       if not visited[s] then
12:         if dfs(s) then
13:           return true
14:     return false
15:   return dfs(Parent(m))
```

Fig. 3: Pseudocode for finding cheeky mallocs using depth-first search

References

[1] Bruno Blanchet. "Escape analysis for Java: Theory and practice". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 25.6 (2003), pp. 713–775.

[2] Lukas Stadler, Thomas Würthinger, and Hanspeter Mössenböck. "Partial escape analysis and scalar replacement for Java". In: *CGO*. 2014.

Results

Benchmarks We implemented three programs: Fibonacci number calculation (`fib`), merge sort (`msort`) and matrix multiplication (`mult`).

	O1	O1 + heap-to-stack	SU	O3	heap-to-stack + O3	SU
fib	0.043	0.020	2.15	0.038	0.016	2.375
msort	19.497	10.530	1.85	19.557	10.548	1.85
mult	0.320	0.318	0.99	0.374	< 0.001	> 100

Fig. 4: Performance on three benchmarks with different allocation patterns (time in seconds)

- Comparison between `O1` and `O1 + heap-to-stack` illustrates the benefits of our optimisation, in isolation
- Comparison between `O3` and `heap-to-stack + O3` illustrates the impact of further optimisations that additionally `heap-to-stack` reveals to `O3`

Performance results

- The speedup shown by `fib` and `msort` demonstrate that moving heap allocations to the stack can lead to a significant speedup
- The speedup obtained by `mult` is owed to the fact that after removing the `malloc` call, the optimizer realized that it could remove a large amount of unnecessary computation. It could not realize this when the `malloc` was present.

Limitations and future work

Allocations inside loops

- Our current implementation does not remove mallocs inside loops.
- This is because the stack has to be carefully popped on each iteration manually, or the allocation must be hoisted outside the loop (à la LICM), which is difficult to get right

Dynamic Promotion

- Our current implementation does not remove mallocs that allocate a variable number of bytes.
- This would involve deferring the decision to allocate on stack to runtime, or performing additional static analysis to find dynamic allocations that are guaranteed to be small
- Careful checks need to be placed to ensure that dynamic promotion does not lead to stack overflow.

Related Work

- Capture tracking and escape analysis have been used somewhat extensively in the context of languages that make frequent use of the heap, such as Java [1]
- Würthinger [2] develop a path-sensitive version of escape analysis, called partial escape analysis, which allows more optimizations to be performed in cases where an allocation may escape in one branch but not another