CS 15-745 Final Report: Promoting Heap Allocations to the Stack in LLVM

Daniel Anderson
dlanders@cs.cmu.edu

Jatin Arora
jatina@cs.cmu.edu

Abstract

Frequent heap allocations can become a performance bottleneck in many programs. Heap-allocated memory is ideal for use in situations when the *lifetime* of the memory is unknown at the time of creation. In some cases however, 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. In such a situation, space providing, the heap-allocated memory could instead be allocated on the stack, which removes the need for system calls to `malloc` and `free`, and thus may provide a speedup. While these kinds of optimizations have been studied a lot for heap-heavy garbage-collected languages like Java, they have not seen much work in manually memory-managed languages like `C` and `C++`.

In this project, we implement an optimization pass in LLVM that identifies `malloc` calls that can be promoted to stack allocations, and perform an emprical evaluation of its performance benefits. We show that on several examples, the transformation can both provide a significant speedup, and also expose opportunities for further optimizations that the optimizer would not have found otherwise.

1 Introduction

Software code typically has access to two different kinds of memory, stack memory and heap memory. The main considerations when deciding between the use of stack and heap memory are (1) the *lifetime* of the object that will be stored in that memory, and (2) the amount of memory that is needed. The lifetime of an object refers to the points in time between its creation and its destruction. In order to store an object on the stack, its lifetime must be *static*, i.e. known in advance. When the lifetime of an object is static, it can be tied to a particular scope of the program (a function call, a loop, a conditional statement, etc.), and hence stored on the stack. When the lifetime of an object is unknown, or it is too large to fit on the stack, the heap, or *dynamically allocated memory* is used instead. Allocating and accessing memory on the stack is typically faster than for heap memory since stack memory does not suffer from fragmentation, and typically offers better locality, so it should be preferred whenever possible.

In low-level languages like C, programmers explicitly express their intention to use heap memory by using `malloc` when they need to acquire memory, and `free`, when it is no longer needed. In some situations, however, a programmer may use dynamic memory, perhaps unintentionally via the use of a standard library data structure, in a situation where static memory would have sufficed. In this project, we aim to design and implement an

optimization pass that can detect some of these situations, i.e. dynamically allocated objects whose lifetime can actually be deduced to be static, and optimize them by transforming the corresponding heap allocations to use stack memory instead.

To solve this problem, we make use of two ingredients. The first is *capture tracking / escape analysis*. Given a pointer p, we say that p is *captured* by a function if that function makes a copy of p (or any part of p) that outlives the call. If a pointer is captured, then we can not, without significant further analysis, decide that the memory it owns has a static lifetime. On the other hand, if a pointer is not captured, and it is guaranteed to be freed in the scope that allocated it, we can deduce that it has a static lifetime. The second ingredient that we need is therefore a way to determine whether an allocation is guaranteed to be freed in the scope that it is allocated. We design an algorithm to solve this part of the problem in Section 3, and analyse its correctness. We then implement our algorithm as a pass in LLVM 10 and perform an empirical evaluation of its performance. We show that for certain programs, promoting heap allocations to the stack results in significant performance benefits.

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]. Since Java does not support allocating objects on the stack, having the compiler decide to do it is important to achieve good performance. Several improvements have also been proposed. For example, since escape analysis is typically very expensive, Vivien and Rinard [4] develop an incremental version that, instead of performing whole-program analysis, incrementally analyzes parts of the program that are deemed likely to be successful. This provides most of the benefits of a whole-program analysis at a fraction of the cost. Stadler and Würthinger [3] 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. Salcianu and Rinard [2] propose a version of escape analysis for multithreaded programs.

2 Preliminaries

Capture Tracking LLVM comes with a simple implementation of capture tracking that, during an analysis pass, determines for each pointer variable, whether it can be deduced to be "nocapture". At a high level, capture tracking is implemented by using a worklist algorithm to traverse the value's uses and determine whether any of them result in capture. For example, calling a function that is read only, has no return value, and does not throw will never capture, but calling a function that returns a value might (because it could return part of the pointer). Calling a function that throws is assumed to capture since the stack unwind could leak the value of the pointer. Similarly, dereferencing a pointer is fine and does not capture it[1], but storing its value does. Perhaps unintuitively, comparing a pointer value also captures it, because comparisons can be used to deduce the value which could then be stored.

[1] Well, aside from some corner cases, like a pointer that points to its own value

3 Our Algorithm

Our goal is to design and implement an algorithm to find `malloc`s that are safe to promote to stack allocations. We already know that the first ingredient to do this, capture tracking, is provided in LLVM. At first glance, one might be tempted to believe that given a pointer that owns memory allocated by `malloc`, if that pointer is not captured, then it is safe to promote. This is insufficient, though. Consider the following example

```
void f(int* p) {
    free(p);
}

void g(int x) {
    int* p = (int*)malloc(4);
    if (x >= 5) {
        f(p);
    }
    else {
        free(p);
    }
}
```

In this case, capture tracking will not deduce p as captured, because `free` is assumed to be non-capturing. Assuming that f is not inlined by the compiler, promoting the `malloc` to a stack allocation would be a bad idea since f would now try to `free` a memory location on the stack, which is undefined behaviour. In order to safely promote a heap allocation to a stack allocation, we need to guarantee that it is both allocated and deallocated in the same scope, so that we can both promote the allocation, and delete the corresponding deallocations. To formalize this requirement, we define the following.

Definition 1 (Cheeky malloc)**.** *Given a malloc instruction*[2]*, we say that it is cheeky if an exit of the enclosing function can be reached from it without passing through an instruction that frees the allocated memory.*

This definition will be useful because it captures the following properties. Given a non-escaping, cheeky malloc instruction, one of the following two things can happen:

1. The memory allocated by the malloc is freed in a different scope
2. The memory allocated by the malloc is leaked

In the first case, we can not promote the malloc to a stack allocation. In the second case, it is less clear. Technically leaking memory is not undefined behaviour, so it would seem incorrect to optimize this case. However, memory allocations are not considered to be observable side effects, so it is in fact perfectly legal for a compiler to optimize out a memory leak in an otherwise well-behaved program. However, since it is difficult to distinguish between these two cases, our algorithm will ignore cheeky mallocs and not optimize them. Now, we will show that being non-cheeky is a sufficient condition for the optimization to be valid.

[2]Technically malloc is not an instruction but a function call to the malloc function, but we will say malloc instruction for brevity

Theorem 1. *Given a well-defined program, and a non-captured, non-cheeky malloc instruction, replacing it with a stack allocation and deleting all of the instructions that deallocate it results in correct program behaviour.*

Proof. Suppose we replace the given malloc instruction with a stack allocation. Clearly, we must delete all corresponding free instructions in order to avoid invoking undefined behaviour by freeing a stack variable. We must argue that removing these frees does not introduce a memory leak by leaving some other allocation not freed. This would require the pointer to be overwritten, so that the free instruction frees some different memory. However, since the program is in SSA form, the only way for the pointer to be overwritten is by another malloc at the same program point, but this malloc is being promoted to a stack allocation, so it will never leak memory.

It remains now to argue that this transformation does not lead to some remaining free instruction freeing the stack memory now pointed to by p. Since the malloc is not cheeky, it is guaranteed to be freed explicitly on all paths that lead to an exit of the function. If there was another free instruction somewhere that freed this memory, then that memory would be double-freed, which is undefined behaviour. Therefore we can conclude that in any well-defined program, replacing the malloc instruction with a stack allocation results in the same program behaviour. □

Our goal now is to design an algorithm for detecting cheeky mallocs. The problem essentially reduces to that of reachability. A malloc is cheeky, if, from its program point, an exit point can be reached without travelling through a deallocation that frees the malloc. We therefore designed the following algorithm, based on depth-first search

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))
```

The effect of loops One additional consideration that we must make when performing this optimization is to be careful of loops. Consider, for instance, the following code

```
for (int i = 0; i < 1000000; i++) {
  int* x = (int*)malloc(256);
  // Do some things
```

```
  free(x);
}
```

Note that we should not naively perform the promotion in this case, because, unlike in actual C code, when implemented in LLVM IR, allocated heap memory is not automatically deallocated at the end of the allocating scope (because of course the notion of scope is no longer present), but rather at the end of the enclosing function. Therefore, replacing the malloc in this code with a stack allocation would cause the program to allocate 256MB on the stack! This exceeds the default stack limit on most systems and hence would likely cause a crash. Further analysis could be performed in order to find safe malloc promotions inside loops, but we leave this for further work, and hence for now, do not promote any mallocs that are inside a loop.

3.1 Implementation in LLVM

Using the framework we defined above, our goal is to detect malloc instructions that are both non-captured and not cheeky. Any such malloc instruction is a candidate for stack allocation. Of course, not all such mallocs can be promoted to stack allocations, because, for example, they might allocate too much memory and run out of stack space. We therefore, for our prototype implementation, only perform the optimization if the malloc allocates a constant amount of memory of at most 64kB. The core part of the optimization then works as follows:

1. Detect a malloc instruction that is non-captured, not cheeky, allocates a constant amount of memory under the threshold, and is not performed inside a loop
2. Replace the malloc instruction with the LLVM `alloca` instruction, which allocates memory on the stack
3. Use a `getelementptr` (GEP) instruction and a `bitcast` to convert the allocated memory to the correct type
4. Replace all uses of the allocated memory with the newly allocated stack memory
5. Remove all corresponding free instructions

3.2 Example

Let's use the following example to demonstrate. We have the following (very unoptimized) C code for computing the square of a number.

```
int square(int x) {
  void* m = malloc(4);
  int* p = (int*)m;
  *p = x*x;
  int answer = *p;
  free(m);
  return answer;
}
```

The IR generated by Clang, after applying `mem2reg` is

```
define dso_local i32 @square(i32 %0) #0 {
  %2 = call noalias i8* @malloc(i64 4) #2
  %3 = bitcast i8* %2 to i32*
  %4 = mul nsw i32 %0, %0
  store i32 %4, i32* %3, align 4
  %5 = load i32, i32* %3, align 4
  %6 = bitcast i32* %3 to i8*
  call void @free(i8* %6) #2
  ret i32 %5
}
```

Running our optimization pass results in the following optimized code

```
define dso_local i32 @square(i32 %0) #0 {
  %2 = alloca [4 x i8], i64 1
  %3 = getelementptr inbounds [4 x i8], [4 x i8]* %2, i64 0
  %4 = bitcast [4 x i8]* %3 to i8*
  %5 = bitcast i8* %4 to i32*
  %6 = mul nsw i32 %0, %0
  store i32 %6, i32* %5, align 4
  %7 = load i32, i32* %5, align 4
  ret i32 %7
}
```

No more malloc! The code is not fully optimized, of course, as there is a redundant bitcast, but these can be cleaned up by further passes (for instance, by running `O3`, although in this example it will simply nuke the entire function and replace it by `return x*x`). To make this example more useful, we omitted to run heavier optimizations before running our pass. This prevents LLVM from simply deleting the entire square function and replacing it with `return x*x`. In our actual benchmark examples, we always run `O1` over the code before applying our optimization, since this makes it work in more cases where it wouldn't otherwise, for example, because a pointer might be copied and aliased, in which case our algorithm would not detect it. `O1` will usually fix this and ensure they are the same value.

4 Experimental Evaluation

Experimental setup We ran experiments on a desktop machine equipped with an Intel Xeon W-2123 at 3.6 GHz, and 16GB of DDR4 main memory, running Ubuntu 16.04. All of our benchmarks implement some algorithms and run them repeatedly (up to one million times) to get more accurate measurements and expose the performance benefits. To see the exact execution counts, refer to the full source code provided with the project. To illustrate the benefits of our optimization, we compile all our benchmarks with four different combinations of optimisation passes.

1. `O1`
2. `O1` + `heap-to-stack`
3. `O3`

4. `O1` + `heap-to-stack` + `O3`

Each of these is composed of 3 passes: `O1`, `O3` and `heap-to-stack` (our optimization pass). For any meaningful impact, the `heap-to-stack` pass requires that capture analysis be done before it executes. Since the `O1` pass performs capture analysis, we run it before running our optimisation. The comparison between `O1` and `O1` + `heap-to-stack` is useful for measuring the performance benefits of our optimisation, in isolation. The `O3` pass is then run on both of these. The comparison between the last two combinations can be used to gauge the impact of further optimisations that `heap-to-stack` exposes for `O3`.

Benchmarks We implemented three applications that our optimization pass had the opportunity to optimize.

1. A recursive computation of Fibonacci numbers (`fib`)
2. Matrix multiplication (`mult`)
3. Bottom-up (non-recursive) merge sort (`msort`)

These simple benchmarks represent a diverse range of situations: recursion, iteration with repeated access, and divide and conquer with new allocations at each level. We provide the implementations of each of these benchmarks in the appendix.

5 Results

The results of our experiments are depicted in Table 1. We show the runtime for each of the combinations of optimization passes, and the corresponding speedup that our optimization pass achieves.

	`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

Table 1: Runtimes and speedups for the three benchmark algorithms (in seconds)

We observe that for both `fib` and `msort`, the speedup achieved is significant before and after running `O3`. In both cases, the speedups are approximately a factor of two. The `mult` benchmark gave very interesting results, although they are slightly misleading so we will elaborate on them more. The relative lack of speedup without `O3` shows that the `heap-to-stack` pass has not actually provided any performance benefit by promoting `mallocs` in this particular example. The reason for the enumerable speedup after running `O3` is actually because the `heap-to-stack` pass exposed an opportunity for `O3` to optimize that it did not see without it. By removing the `malloc`, the optimizer now realized that the loop iterations of some of the loops in the program all produced exactly the same outcome each time, and hence optimized them to only run them once. It did not find this optimization when `malloc` was present. These benchmarks therefore reveal two advantages of promoting `mallocs` to stack allocations, which we elaborate on below.

Allocating on the stack is more efficient when possible The most obvious benefit of promoting `mallocs` to stack allocations is that they are significantly cheaper. While `malloc` is required to implement an algorithm to find and reuse suitable chunks of memory, recycling them whenever they are deallocated, performing a stack allocation simply requires moving the stack pointer. This has several advantages:

1. Fewer instructions: Moving the stack pointer is a single instruction, rather than the many required to perform `malloc` and `free`
2. Less memory fragmentation: Since the stack is always contiguous, memory will not be fragmented. When allocating on the heap, sequential allocations are not guaranteed to be adjacent in memory. This leads to the following point.
3. Better locality: Since stack allocations are all adjacent, they experience better locality and hence perform better due to caching.

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 and this discourages many optimisations from relying on them.

6 Limitations

Our current implementation has the following limitations. Some are discussed in more detail as potential avenues for future work momentarily.

- It does not remove mallocs inside for loops.
- The `C++` standard library uses a special function called `operator new` to allocate memory. By default, this simply defers to malloc, but because it is an operator, is overridable. This means that any code using operator new can have arbitrary side effects that are not knowable until link time. For example, it is common for profiling purposes to instrument the operator to count the total amount of memory that has been allocated by the program. This means that our optimization is less effective on `C++` code than on C code. However, there are parts of the `C++` standard that suggest that removing memory allocations is a valid optimization even if they would have otherwise produced observable side effects.
- Our current analysis does not remove mallocs that allocate a variable number of bytes. Furthermore, we did not establish a specific threshold for the size of allocations that are good targets for promotion.

7 Surprises

- The capture analysis done by LLVM handles a very limited form of recursion. The analysis considers that a pointer escapes a function if it makes two or more recursive calls. If it makes only one recursive call, however, no capture occurs. Theoretically, there does not seem to be any reason to consider the number of recursive calls for capture tracking.

- Adding `alloca` instructions to a program after the tail call optimisation has been done leads to an unsound program. It is a fair assumption that the code for de-allocating stack frames be created during the code generation for assembly, especially since there are no instructions for popping the stack in the intermediate representation (except for popping explicit allocations done by `alloca`). Therefore, we were confused as to why our code was crashing after adding stack allocations. It took a lot of time to discover that it is not okay to add them after tail call optimisation, but must be done before. Our optimization pass must therefore be ran before `O3` is run, not after.

8 Conclusions

In this project, we explored the benefits of promoting `mallocs` to stack allocations. We gave an algorithm for finding `mallocs` that are safe candidates for promotion, and performed an empirical study which showed that on certain programs, it results in a significant performance benefit. This project opens up many exciting avenues to explore for future work, which we discuss below.

Future work There are several interesting and natural extensions to this work. An interesting question is how to most effectively handle mallocs that occur inside loops. As we discussed earlier, naively promoting mallocs that are inside loops could lead to blowing the stack, but further analysis could be performed to find safe ways to promote. We see two main ways that this could be achieved

1. When promoting a malloc inside a loop, find a safe place to subsequently pop the allocated stack memory before the loop iterates again. This requires being very careful about the order of allocations to make sure that nothing is unintentionally popped off the stack that shouldn't be

2. Hoist the allocation outside of the loop (à la LICM). Careful considerations need to be made here in order to avoid hoisting out expensive allocations that might not always occur, as is the case in LICM in general.

Another question that warrants further investigation is handling dynamically sized allocations. In our implementation, we only promote constant size allocations since we do not want to accidentally allocate a huge amount of stack memory and blow the stack. Conditional checks could be used to selectively promote only small allocations, but this comes at the expense of also having to selectively deallocate, which could just decrease the performance of the program if small allocations are rare. An alternative approach could be to use a static analysis to determine if any dynamically sized allocations can be guaranteed to be small, say, under some threshold, even if their precise value is not known until runtime. These could then be promoted to stack allocations unconditionally.

Lastly, it would be useful to explore a larger number of benchmark applications to obtain a more complete picture of precisely what program characteristics make them amenable to this optimization. This would also allow us to find potential instances in which this optimization might result in a performance decrease, and to improve our algorithm to proactively guard against them.

Major changes from proposal / milestone report We bounced back and forth a few times here. Our original plan was to implement our own version of escape analysis and compare it to LLVM's. In our milestone report, we decided to switch gears and look into how to improve escape analysis for recursive functions. We discovered later though, that LLVM's analysis for recursive functions was already sufficiently interesting. We then discovered that LLVM does not actually have an implementation of `malloc` to stack promotion at all. We therefore decided that since LLVM already has an implementation of escape analysis, it would be less interesting to compare our own implementation to LLVM's, and instead decided to switch right back to our very first idea, which was to implement an optimization that promotes suitable `malloc` calls to stack allocations.

Distribution of Total Credit: 50, 50

References

[1] B. Blanchet. Escape analysis for Java: Theory and practice. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 25(6):713–775, 2003.

[2] A. Salcianu and M. Rinard. Pointer and escape analysis for multithreaded programs. *ACM SIGPLAN Notices*, 36(7):12–23, 2001.

[3] L. Stadler, T. Würthinger, and H. Mössenböck. Partial escape analysis and scalar replacement for java. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 165–174, 2014.

[4] F. Vivien and M. Rinard. Incrementalized pointer and escape analysis. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 35–46, 2001.

Appendix: Benchmark Code

Fibonacci numbers

```
// Computes a Fibonacci number up to Fib(100)
#define N 100
int f(int n) {
  int* buffer = (int*) malloc(N * sizeof(int));
  buffer[0] = buffer[1] = 1;
  for (int i = 2; i < n; i++)
    buffer[i] = buffer[i-1] + buffer[i-2];
  int ans = buffer[n-1];
  free(buffer);
  return ans;
}
```

Matrix multiplication

```
// Multiplies two matrices and computes the sum of the elements of the result.
#define N 100
int multiply(int n) {
  int* A = (int*) malloc(N*N*sizeof(int));;
  int* B = (int*) malloc(N*N*sizeof(int));
  int* res = (int*) malloc(N*N*sizeof(int));

  for(int i=0; i<N; i++){
    for(int j=0; j<N; j++){
      int offset = i*N+j;
      A[offset]= i+j;
      B[offset] = i-j;
    }
  }

  for (int i = 0; i < N; i++){
    for (int j = 0; j < N; j++){
      res[i*N+j] = 0;
      for (int k = 0; k < N; k++)
        res[i*N+j] += A[i*N + k]*B[k*N + j];
    }
  }

  int ans = 0;
  for(int i=0; i<N; i++)
    for(int j=0; j<N; j++)
      ans+=res[i*N+j];

  free(A); free(B); free(res);
  return ans;
}
```

Merge sort

```
#define N 10000
void merge(int* arr, int from, int mid, int to) {
  int* sorted = (int*) malloc(N*sizeof(int));

  int idxl = from, idxr = mid+1;
  int idx = from;
  while(idxl <= mid && idxr<=to) {
    if(arr[idxl] < arr[idxr]){
      sorted[idx++] = arr[idxl++];
    }
    else {
      sorted[idx++] = arr[idxr++];
    }
  }

  while(idxl<= mid && idxl<N) {
    sorted[idx++] = arr[idxl++];
  }

  for(int i= from; i<=to; i++)
    arr[i] = sorted[i];
  free(sorted);
}

// Sort the given array
void iterativeMergeSort(int *arr, int len) {
  for(int d = 1; d < len; d*=2) {
    for(int grain = 0; grain < len - 1; grain+=(2*d)) {
      int from = grain;
      int mid = grain + d-1;
      int to = ((mid+d)>(len-1))?(len-1):(mid+d);
      merge(arr, from, mid, to);
    }
  }
}

void f() {
  int* p = (int*)malloc(16*sizeof(int));
  p[0] = 1;
  for (int i = 0; i < 16; i++) {
    p[i] = p[i-1]
  }
  free(p);
}
```