From Heap to Stack

Moving heap-allocated objects to the stack in LLVM

Daniel Anderon
dlanders@cs.cmu.edu

Jatin Arora
jatina.cs.cmu.edu

March 2020

1 Project Information

Project Title From Heap to Stack – Moving heap-allocated objects to the stack in LLVM

Group Info Daniel Anderson (dlanders) and Jatin Arora (jatina)

URL For the Project Web Page Project URL

Project Description 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.

Ideas like this have been studied under the context of *escape analysis*. Escape Analysis is a static analysis that determines if the lifetime of the data exceeds it's static scope. If it is the case, then the object can be allocated on the stack despite the programmer's specification of where it should be stored. In this project, we plan to implement this heap-to-stack transformation in the LLVM compiler infrastructure and benchmark it on programs written in C++.

- 75% Goal: Provide an empirical study of the performance improvements that can be achieved by using escape analysis (possibly using LLVM's existing escape analysis if ours does not work)
- 100% Goal: Implement a fully functional escape analysis and heap-to-stack transformation and benchmark its performance against existing implementations, providing an empirical study of the benefits.
- 125% Goal: Implement a fully functional escape analysis and benchmark its performance against existing implementations. Demonstrate theoretically or experimentally that our implementation improves over the performance of existing methods.
- 200% Goal: Investigate and implement other applications of escape analysis, such as eliminating unnecessary synchronization in concurrent programs in situations where the objects in question never escape their allocating scope.

2 Logistics

Schedule

1. (Week 1: 23rd March): Submit Project Proposal (Both)
2. (Week 2: 30th March): Investigate escape analysis literature and existing work (Both)
3. (Week 3: 6th April): Get familiar with LLVM's implementation of escape analysis and have an algorithm to implement (Both)
4. (Week 4: 13th April): Implement the algorithm (Both)
5. (Week 5: 20th April): Implement the algorithm and benchmark the performance (Both)
6. (Week 6: 27th April): Project due on the 29th April (Both)

The bottleneck is likely to be coming up with possible improvements to existing algorithms. If we fail to do this, we will implement some existing methods and compare them. We will work collaboratively throughout the project. The main points at which we will divide work are when implementing the different parts of the algorithm (Week 4 and 5) and when implementing and analysing the results of benchmarks (Week 5). We will do the division once we are familiar enough with the algorithms to decide.

Milestone By April 16th, we plan to have designed an algorithm that we want to implement and to have begun implementing it (see schedule Weeks 1—4).

Literature Search

1. Escape Analysis for Java
2. Incrementalized Pointer and Escape Analysis
3. Implementing Andersen Alias Analysis in LLVM
4. Pointer and escape analysis for multithreaded programs

Resources Needed We anticipate the need of the following resources:

1. A computer
2. LLVM compiler infrastructure
3. ACM digital library

We will use the latest stable release of the LLVM infrastructure, which is LLVM 9 at the time of writing (`https://apt.llvm.org/`).

Getting started: Constraints preventing you from getting started immediately

1. Being stuck in another city due to coronavirus-related travel restrictions.