Task 4
GitHub

1 Introduction

The current task is to familiarize ourselves with LLVM optimization techniques. The following report describes the implementation of two optimizations. The first task, dead code elimination, is a LLVM pass that is performed as a part of static analysis during compile time. The second task, memory bounds checking, performs code instrumentation and optimization to self-detect errors during execution dynamically.

2 Dead Code Elimination

Dead Code Elimination (DCE) is a LLVM pass that finds and removes dead code from the LLVM intermediate representation. Dead code consists of instructions that are executed (or not), but whose result is never used in further computations.

2.1 Setup and Methodology

In order to run the process we have included a script `start.sh` in the root directory. We clone the assignment files from GitHub, then inside of it clone llvm-project. We then proceed to copy our modified files to the correct locations under LLVM, build it and then the `opt` tool as well. LLVM contains the `opt` tool which is very convenient for testing optimizations and analyses. In our case we provide a file that contains a function with dead code in intermediate representation and verify that our DCE implementation returns the function without the dead code. The exact command we use is `./build/bin/opt -S -passes=dead-code-elimination-pass FunctionWithDeadCode.ll`. We've passed the `-S` flag in order to get the optimized function code, and witness that the dead code there was removed.

The file containing the pass implementation is `DeadCodeElimination.cpp` and the corresponding header is `DeadCodeElimination.h`. However, we first have to modify a couple other files in order for our pass to be compiled into LLVM and available in the `opt` tool. You can find the instructions in this README file.

2.2 Implementation

The Dead Code Elimination pass receives a `Function` object and a `FunctionAnalysisManager` as parameters. It iterates over all instructions in the function and uses the [3] function to identify trivially dead code. The function `isInstructionTriviallyDead` checks if the instruction's return value has any use, and if the instruction has any side effects (e.g. printing to stdout, calling another non-dead function). Because objects should not be removed from the list while iterating, a worklist is used to store the intructions that should be removed. Before removing the instruction with the `eraseFromParent` function, we iterate over the operands of the instruction and if they become trivially dead instructions when nulled we add them to the worklist. This is necessary because for example when a function `foo`'s return value is used by only one other function `bar`, whose return value is never used, the first function `foo` is considered alive because its value is used, however the only function which uses that value is a dead function. We thus have to remove `bar` first, iterate over the function's instructions again to see if some died as a consequence of that, and remove them.

Once we delete an instruction `I` from the code another instruction that was used by `I` may become dead as well. To handle all such cases we use the `changed` flag which stores whether any instruction was removed during an iteration as described above. If so, we repeat, otherwise there is no trivially dead code anymore.

3 Aggressive Dead Code Elimination

ADCE, similar to DCE, but assumes all instructions to be dead unless proven otherwise, i.e. it adds add all instructions to the worklist and then performs liveliness checks on them. The difference between the two techniques can be explained with conditionally dead code. That is, code that is being run depending on special conditions such as OS version, installed drivers, packages, and so on. When there is doubt DCE doesn't remove dead code, in comparison to ADCE that does.

4 Bounds Checker

4.1 Setup and Methodology

Instead of building from source, in Assignment 2, we use the binary packages provided by nix directly. After initiating the nix environment by `nix-shell`, the script `start.sh` will automatically clone the template repository into the directory `llvm-template`, and copy the modified Makefiles and source codes to the correct places. The project is then compiled and the target application `mytest` will run.

There are two independent passes for Assignment2.1 and Assignment2.2 respectively, namely `boundschecker` and `boundschecker_opt`. Each pass works upon the test codes directly. The pass `boundschecker` will replace malloc call with `__runtime_mymalloc`, and do IR Instrumentation to insert a function call `__runtime_checkbound()` before instructions load and store. The runtime library uses a simplified Address Sanitizer method to do memory bound check. The pass `boundschecker_opt` is a reimplementation of the original boundschecker with more checks and control on flow in order to remove redundancy access to memory.

As LLVM provides a hook to allow plugin modules to be run automatically when loaded by clang, this lets us use clang and our pass(`-Xclang -load -Xclang $(PASS_SharedLibrary)`) as a drop-in replacement for the normal C compiler. The target program is called `mytest` and `mytest_opt` respectively. And the Intermediate Representation code (`.bc` and `.ll`) will also be generated with tool `opt` and `llvm-dis` in our Makefile. See and compare the final result in the directory `llvm-template/Assignment2/tests/` with the file name `assignment2.1.ll` and `assignment2.2.ll`.

4.2 Result of Boundschecker

Note that for illustrating the effect of removing redundant boundcheckers in the exploration task, we have changed the main.c file in order to reflect slides illustration on value store behavior. The following code snippet is the running result of the program `mytest`.

```
> assign_malloc. 0: 0
> assign_malloc. 1: 10
> assign_malloc. 2: 20
> assign_malloc. 3: 30
> assign_malloc. 4: 40
>>>>>>>>>> Out-of-bounds error: 0xba30fb12b4
> assign_malloc. 5: 50
> main -- alloc[3]: 30
> main -- alloc[4]: 40
> main-- alloc[4]: 20 <--- no buffer
   overflow, but reassign a new value
>>>>>>>>>> Out-of-bounds error: 0xba30fb12f8
> main-- alloc+22: 1041 <--- buffer overflow
```

We can see that, the warning message (`Out-of-bounds error: xxx`) will trigger when we try to access invalid memory address, e.g. store to alloc[5] and load from alloc[22].

Lets prove the IR code now. It's obvious that the malloc call has been replaced and there are bounds checker before load and store instructions.

```
%5 = shl nuw nsw i64 %4, 2
%6 = call noalias i8* @__runtime_mymalloc(
    i64 %5) #4
%7 = add i32 %3, 1
...
%10 = bitcast i32* %9 to i8*
%11 = call i8* @__runtime_checkbound(i8*
    %10)
%12 = bitcast i8* %11 to i32*
%13 = load i32, i32* %12, align 4, !tbaa
    !5
%14 = call i32 (i32, i8*, ...)
    @__printf_chk(i32 1, i8* getelementptr
     inbounds ([24 x i8], [24 x i8]* @.str
    , i64 0, i64 0), i32 %13) #4
```

4.3 Implementation

Both boundschecker solution has been implementated as *ModulePass*. As in the starting implementation, we do check and dynamically cast *LoadInst* and *StoreInst* with a pointer operand that's different from a constant. New implementation avoids double checks on the same address keeping track of already called memory pointer using a *SmallVector <Value*, 0> PointerOperandList*. If the address we're loading / storing has already been instrumented in a previous call, this will automatically be skipped as we can assume it is safe, with the help of a boolean flag that is set as true if a *Value** with an already present in the list address is found. In the following example, we can assert that the store instruction has been called without any previous check as the pointer as already been proven to be valid in the previous *runtimeCheckbound* call.

```
%18 = call i8* @__runtime_checkbound(i8*
    %17)
%19 = bitcast i8* %18 to i32*
%20 = load i32, i32* %19, align 4, !tbaa
    !5
%21 = call i32 (i32, i8*, ...)
    @__printf_chk(i32 1, i8* getelementptr
     inbounds ([24 x i8], [24 x i8]* @.str
    .1, i64 0, i64 0), i32 %20) #4
store i32 20, i32* %16, align 4, !tbaa !5
```

References

[1] https://www.inf.ed.ac.uk/teaching/courses/ct/19-20/slides/llvm-4-deadcode.pdf

[2] https://learning.oreilly.com/library/view/llvm-cookbook/9781785285981/ch02s12.htmlch02lvl2sec90

[3] https://docs.hdoc.io/hdoc/llvm-project/f9F06E8E56FA5B377.html

[4] https://llvm.org/docs/WritingAnLLVMNewPMPass.html