Lab 4: Bounds Check Analysis

Bert De Saffel, Xandro Vermeulen

May 20, 2019

Contents

1 Introduction

This report describes the performance of three different `IR` representations of a single `C` program, `bubble_sort.c`. The first `IR` representation is the standard representation without a function pass. The second implementation modifies the `IR` by applying bound checking using branch instructions for each `getelementptr` instruction. The third representation also applies bound checking, but with an extra optimalisation:

2 Modification of the IR

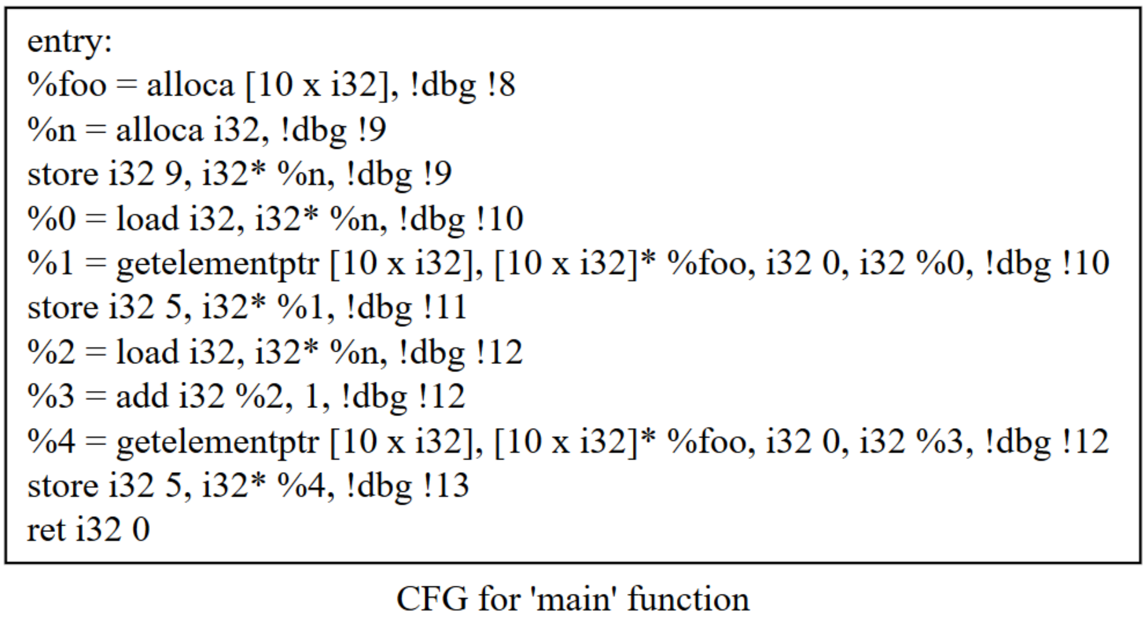
When implementing runtime bounds checking, transformations to the IR have to be made as the default IR does not handle runtime bounds checking. To demonstrate the changes, a simple *C* program is used:

```
int foo[10];
int n = 9;
foo[n] = 5;
foo[n + 1] = 5;
```

The second array indexing on line 4 should result in a runtime error. The original and modified IR are shown respectively on figure 1 and 2. The IR on figure 1 will simply give a segmentation fault when line 4 is executed. To combat this, each `getelementptr` instruction is preceded by a `icmp` instruction, which compares the index value with the array size, and a `br` instruction which will chose the `trap` or `cont` block based on the resulting boolean value of the `icmp` instruction. The resulting IR is shown on figure 2.

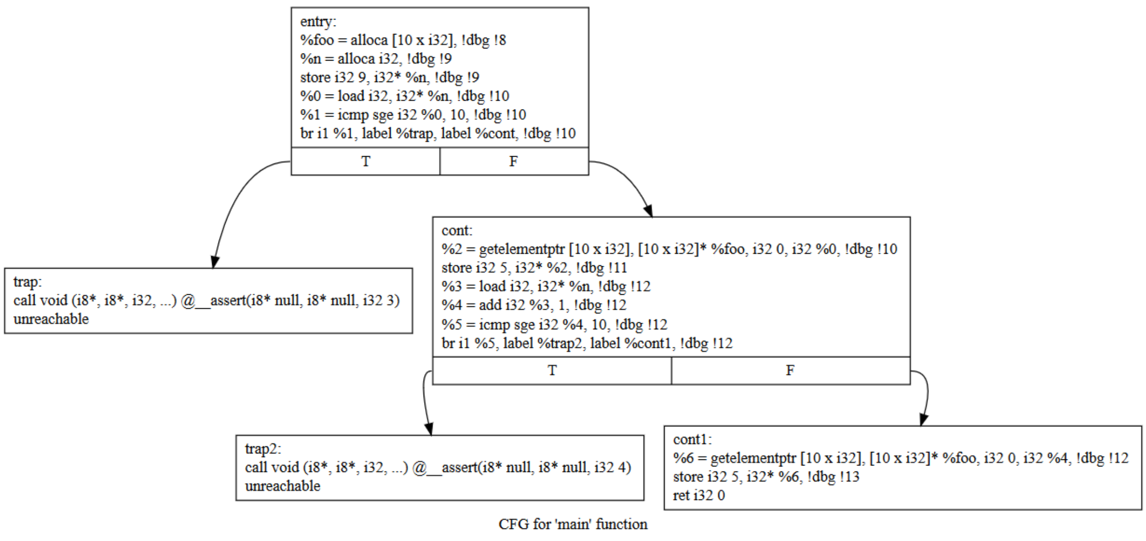
3 Analysis

Using the method above, a program that contains n `getelementptr` instructions will contain $2n$ extra basic blocks, n extra `icmp` instructions and n extra `br` instructions. To measure the impact of these extra instructions and basic blocks, the program `bubble_sort.c` is executed 30 times for both representations. Afterwards, average statistics such as runtime can be calculated for each representation. It can be assumed that inserting more code to perform the runtime bounds checking



```
entry:
%foo = alloca [10 x i32], !dbg !8
%n = alloca i32, !dbg !9
store i32 9, i32* %n, !dbg !9
%0 = load i32, i32* %n, !dbg !10
%1 = getelementptr [10 x i32], [10 x i32]* %foo, i32 0, i32 %0, !dbg !10
store i32 5, i32* %1, !dbg !11
%2 = load i32, i32* %n, !dbg !12
%3 = add i32 %2, 1, !dbg !12
%4 = getelementptr [10 x i32], [10 x i32]* %foo, i32 0, i32 %3, !dbg !12
store i32 5, i32* %4, !dbg !13
ret i32 0
```

Figure 1: The original, unmodified IR for program REF.



```
entry:
%foo = alloca [10 x i32], !dbg !8
%n = alloca i32, !dbg !9
store i32 9, i32* %n, !dbg !9
%0 = load i32, i32* %n, !dbg !10
%1 = icmp sge i32 %0, 10, !dbg !10
br i1 %1, label %trap, label %cont, !dbg !10
```

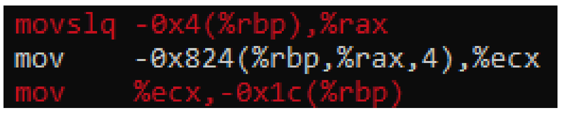
```
trap:
call void (i8*, i8*, i32, ...) @__assert(i8* null, i8* null, i32 3)
unreachable
```

```
cont:
%2 = getelementptr [10 x i32], [10 x i32]* %foo, i32 0, i32 %0, !dbg !10
store i32 5, i32* %2, !dbg !11
%3 = load i32, i32* %n, !dbg !12
%4 = add i32 %3, 1, !dbg !12
%5 = icmp sge i32 %4, 10, !dbg !12
br i1 %5, label %trap2, label %cont1, !dbg !12
```

```
trap2:
call void (i8*, i8*, i32, ...) @__assert(i8* null, i8* null, i32 4)
unreachable
```

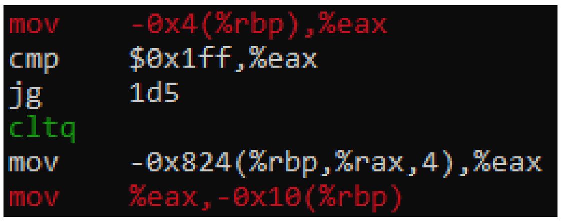
```
cont1:
%6 = getelementptr [10 x i32], [10 x i32]* %foo, i32 0, i32 %4, !dbg !12
store i32 5, i32* %6, !dbg !13
ret i32 0
```

Figure 2: Modified IR for program REF.



```
movslq -0x4(%rbp),%rax
mov    -0x824(%rbp,%rax,4),%ecx
mov    %ecx,-0x1c(%rbp)
```

(a) Generated assembly code for the original IR.



```
mov    -0x4(%rbp),%eax
cmp    $0x1ff,%eax
jg     1d5
cltq
mov    -0x824(%rbp,%rax,4),%eax
mov    %eax,-0x10(%rbp)
```

(b) Generated assembly code for the modified IR.

Figure 3: x86-64 assembly code for line 33 in `bubble_sort.c`.

will increase performance overhead. Indeed, the average runtime for the original and modified representation is respectively `2758 ms` and `3141 ms`, resulting in a 13% increase for the modified version. This has a straightforward explanation: each `getelementptr` is preceded by a `icmp` and `br` instruction. To explain this, we consider line 33 of `bubble_sort.c`, which contains the following code:

```
a = numbers[j]
```

The generated assembly code is shown on figure 3a. This shows how the value of j, which resides at an offset of 4 bytes from the base pointer, is put in the `%RAX` register. Then the array `numbers` is accessed by calculating the correct address and putting the value which resides at this address in the `%ECX` register. This value is then put into the variable a, which resides at an offset of 28 ($0x1c$) bytes from the base pointer. Now compare this to figure 3b, which contains 3 new instructions as a result of the runtime bounds check. The first instruction is a `cmp` instruction, which compares the maximum array offset (0x1ff = 511) to the index value (the j variable). The second instruction is a `jg` instruction, which jumps to its corresponding `trap` block if the previous `cmp` instruction holds true ($j > 511$). A third instruction is `cltq`, which converts the value in the `EAX` register from a long (4 bytes) to a quad (8 bytes). This conversion is neccesary because the register `%EAX` is a 32-bit register and the following operations use 64-bit registers. The word length of each register in an instruction must be equal. Of these three new instructions, the `cltq` instruction takes the most time.

However, this could be avoided if a 64-bit integer type was used in the IR instead of a 32-bit integer type. Unfortunately, this requires that the source program also uses unsigned integers.

3.1 Optimalisation