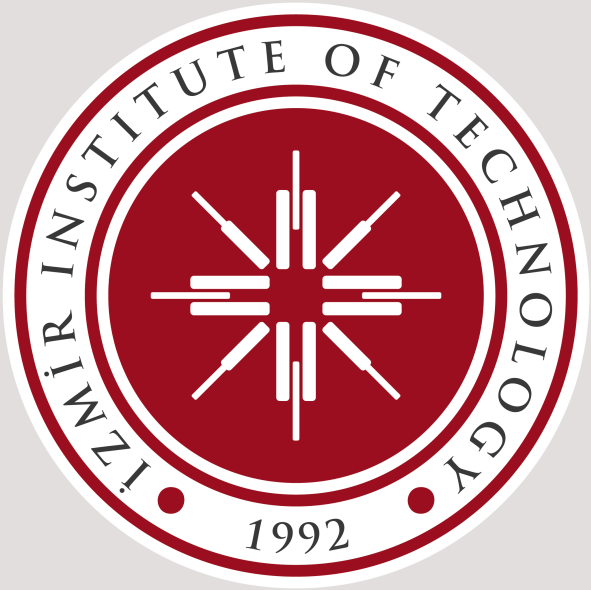
Izmir Institute of Technology
Computer Engineering Department
CENG513 Final Exam Spring 2024
Question 1

Student Name: Gökay Gülsoy Student No: 270201072

June 11, 2024



Question 1

I have chosen following three LLVM transformation passes to be applied on simple C codes written by myself:

1. Dead store elimination (dse)
2. Function inlining (inline)
3. Loop invariant code motion (licm)

Dead store elimination **dse** transformation pass locates redundant local store operations inside a basic block and eliminates them [1]. Sequence of commands to be executed in order to generate unoptimized LLVM IR file and after that applying dead store elimination transformation pass to it for generating optimized IR file is as follows:

1. clang -O0 -emit-llvm -S -Xclang -disable-O0-optnone dead_code.c -o dead_code_IR.ll
2. opt -passes=dse dead_code_IR.ll -S -o dead_code_eliminated_IR.ll
3. opt -passes=dot-cfg dead_code_IR.ll
4. dot -Tsvg dead_code_IR.dot -o dead_code_cfg.svg
5. opt -passes=dot-cfg dead_code_eliminated_IR.ll
6. dot -Tsvg dead_code_IR.dot -o dead_code_eliminated_cfg.svg

first command generates an IR file by turning off optimizations. Second command runs the transformation pass given by the **-passes=pass_name** flag, in that case dse stands for dead store elimination which is used by LLVM to specify the type of transformation [2]. Third command generates .dot files for unoptimized IR file in order to generate CFG. Fourth command takes a .dot file for unoptimized IR file and generates a visualization of CFG as a .svg file. Fifth and sixth steps repeat the generation of .dot files and CFG for optimized code. Following figures show the dead_code.c source code file which contains dead code inside the function named doNothing and variable named unused inside the main function. Running dead store elimination (dse) pass transforms the IR given in figure 2 by removing store operation inside the doNothing function and by removing store operation inside the main function. Optimized IR file generated as a result of dead store elimination pass is given in the figure 3.

File: dead_code.c

```c
#include <stdio.h>

void doNothing() {
    int i = 10; // This assignment is dead code
}

int main() {
    doNothing(); // The call to doNothing is effectively dead code

    int a = 10;
    int b = 20;
    int result = a + b;
    printf("Result: %d\n", result);

    int unused = 100; // This variable is never used

    return 0;
}

```

Figure 1: C Source code file containing dead code

File: dead_code_IR.ll

```
; ModuleID = 'dead_code.c'
source_filename = "dead_code.c"
target datalayout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-i128:128-f80:128-n8:16:32:64-S128"
target triple = "x86_64-unknown-linux-gnu"

@.str = private unnamed_addr constant [12 x i8] c"Result: %d\0A\00", align 1

; Function Attrs: noinline nounwind uwtable
define dso_local void @doNothing() #0 {
  %1 = alloca i32, align 4
  store i32 10, ptr %1, align 4
  ret void
}

; Function Attrs: noinline nounwind uwtable
define dso_local i32 @main() #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  %5 = alloca i32, align 4
  store i32 0, ptr %1, align 4
  call void @doNothing()
  store i32 10, ptr %2, align 4
  store i32 20, ptr %3, align 4
  %6 = load i32, ptr %2, align 4
  %7 = load i32, ptr %3, align 4
  %8 = add nsw i32 %6, %7
  store i32 %8, ptr %4, align 4
  %9 = load i32, ptr %4, align 4
  %10 = call i32 (ptr, ...) @printf(ptr noundef @.str, i32 noundef %9)
  store i32 100, ptr %5, align 4
  ret i32 0
}

```

Figure 2: IR file for unoptimized code

File: dead_code_eliminated_IR.ll

```
; ModuleID = 'dead_code_IR.ll'
source_filename = "dead_code.c"
target datalayout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-i128:128-f80:128-n8:16:32:64-S128"
target triple = "x86_64-unknown-linux-gnu"

@.str = private unnamed_addr constant [12 x i8] c"Result: %d\0A\00", align 1

; Function Attrs: noinline nounwind uwtable
define dso_local void @doNothing() #0 {
  ret void
}

; Function Attrs: noinline nounwind uwtable
define dso_local i32 @main() #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %3 = alloca i32, align 4
  call void @doNothing()
  store i32 10, ptr %1, align 4
  store i32 20, ptr %2, align 4
  %4 = load i32, ptr %1, align 4
  %5 = load i32, ptr %2, align 4
  %6 = add nsw i32 %4, %5
  store i32 %6, ptr %3, align 4
  %7 = load i32, ptr %3, align 4
  %8 = call i32 (ptr, ...) @printf(ptr noundef @.str, i32 noundef %7)
  ret i32 0
}

```

Figure 3: IR file for optimized code

Function inlining transformation pass inserts the whole function body into main function code where a call made to function [3]. Sequence of commands to be executed in order to generate unoptimized LLVM IR file and after that applying function inlining transformation pass to it for generating optimized IR file is as follows:

1. clang -O0 -emit-llvm -S -Xclang -disable-O0-optnone function_inlining.c -o function_inlining_IR.ll
2. opt -passes=inline dead_code_IR.ll -S -o function_inlined_IR.ll
3. opt -passes=dot-cfg function_inlining_IR.ll
4. dot -Tsvg function_inlining_IR.dot -o function_inlining_cfg.svg
5. opt -passes=dot-cfg function_inlined_IR.ll
6. dot -Tsvg function_inlined_IR.dot -o function_inlined_cfg.svg

Figure 4 shows the C source code file for the function inlining example, figure 5 shows the unoptimized IR file for function inlining source code, figure 6 shows the optimized IR file for function inlining source code respectively.

File: function_inlining.c

```c
#include <stdlib.h>
#include <stdio.h>

// simple function to sum
// two integer values
int sum(int a,int b) {
    int c = a + b;
    return c;
}

int main() {
    int a = 15;
    int b = 20;
    int c = 25;

    int e = sum(a,b) + c;

    return 0;
}
```

Figure 4: C Source code file containing uninlined code

File: function_inlining_IR.ll

```
; ModuleID = 'function_inlining.c'
source_filename = "function_inlining.c"
target datalayout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-i128
:128-f80:128-n8:16:32:64-S128"
target triple = "x86_64-unknown-linux-gnu"

; Function Attrs: nounwind uwtable
define dso_local i32 @sum(i32 noundef %0, i32 noundef %1) #0 {
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  %5 = alloca i32, align 4
  store i32 %0, ptr %3, align 4
  store i32 %1, ptr %4, align 4
  %6 = load i32, ptr %3, align 4
  %7 = load i32, ptr %4, align 4
  %8 = add nsw i32 %6, %7
  store i32 %8, ptr %5, align 4
  %9 = load i32, ptr %5, align 4
  ret i32 %9
}

; Function Attrs: nounwind uwtable
define dso_local i32 @main() #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  %5 = alloca i32, align 4
  store i32 0, ptr %1, align 4
  store i32 15, ptr %2, align 4
  store i32 20, ptr %3, align 4
  store i32 25, ptr %4, align 4
  %6 = load i32, ptr %2, align 4
  %7 = load i32, ptr %3, align 4
  %8 = call i32 @sum(i32 noundef %6, i32 noundef %7)
  %9 = load i32, ptr %4, align 4
  %10 = add nsw i32 %8, %9
  store i32 %10, ptr %5, align 4
  ret i32 0
}
```

Figure 5: IR file for unoptimized function_inlining.c

File: function_inlining_IR.ll

```
; ModuleID = 'function_inlining.c'
source_filename = "function_inlining.c"
target datalayout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-i128
:128-f80:128-n8:16:32:64-S128"
target triple = "x86_64-unknown-linux-gnu"

; Function Attrs: nounwind uwtable
define dso_local i32 @sum(i32 noundef %0, i32 noundef %1) #0 {
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  %5 = alloca i32, align 4
  store i32 %0, ptr %3, align 4
  store i32 %1, ptr %4, align 4
  %6 = load i32, ptr %3, align 4
  %7 = load i32, ptr %4, align 4
  %8 = add nsw i32 %6, %7
  store i32 %8, ptr %5, align 4
  %9 = load i32, ptr %5, align 4
  ret i32 %9
}

; Function Attrs: nounwind uwtable
define dso_local i32 @main() #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  %5 = alloca i32, align 4
  store i32 0, ptr %1, align 4
  store i32 15, ptr %2, align 4
  store i32 20, ptr %3, align 4
  store i32 25, ptr %4, align 4
  %6 = load i32, ptr %2, align 4
  %7 = load i32, ptr %3, align 4
  %8 = call i32 @sum(i32 noundef %6, i32 noundef %7)
  %9 = load i32, ptr %4, align 4
  %10 = add nsw i32 %8, %9
  store i32 %10, ptr %5, align 4
  ret i32 0
}
```

Figure 6: IR file for optimized function_inlining.c

Loop invariant code motion transformation pass attempts to remove as much code from the body of the loop as possible via hoisting code into preheader block, or by sinking code to exit blocks. Sequence of commands to be executed in order to generate unoptimized LLVM IR file and after that applying loop invariant code motion transformation pass to it for generating optimized IR file is as follows:

1. clang -O0 -emit-llvm -S -Xclang -disable-O0-optnone loop_invariant_code_motion.c -o loop_invariant_code_motion_IR.ll
2. opt -passes=licm loop_invariant_code_motion_IR.ll -S -o loop_invariant_code_motion_applied_IR.ll
3. opt -passes=dot-cfg loop_invariant_code_motion_IR.ll
4. dot -Tsvg loop_invariant_code_motion_IR.dot -o loop_invariant_code_motion_cfg.svg

5. opt -passes=dot-cfg loop_invariant_code_motion_applied_IR.ll

6. dot -Tsvg loop_invariant_code_motion_applied_IR.dot -o loop_invariant_code_motion_applied_cfg.svg

Figure 7 shows the C source code file for the loop invariant code motion example, figure 8 shows the unoptimized IR file for loop invariant code motion source code, figure 9 shows the optimized IR file for loop invariant code motion source code respectively.

File: loop_invariant_code_motion.c

```c
#include <stdlib.h>
#include <stdio.h>

int main() {
    int a = 10;
    int b = 20;

    int i = 0;
    while (i < 5) {
        int c = a + b; // loop invariant code
        printf("Value of a*b is: %d",a*b);
        i++;
    }

    return 0;
}

```

Figure 7: C Source code file containing loop invariant

File: loop_invariant_code_motion_IR.ll

```
; ModuleID = 'loop_invariant_code_motion.c'
source_filename = "loop_invariant_code_motion.c"
target datalayout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-i128:128-f80:128-n8:16:32:64-S128"
target triple = "x86_64-unknown-linux-gnu"

@.str = private unnamed_addr constant [20 x i8] c"Value of a*b is: %d\00", align 1

; Function Attrs: noinline nounwind uwtable
define dso_local i32 @main() #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  %5 = alloca i32, align 4
  store i32 0, ptr %1, align 4
  store i32 10, ptr %2, align 4
  store i32 20, ptr %3, align 4
  store i32 0, ptr %4, align 4
  br label %6

6:                                                ; preds = %9, %0
  %7 = load i32, ptr %4, align 4
  %8 = icmp slt i32 %7, 5
  br i1 %8, label %9, label %19

9:                                                ; preds = %6
  %10 = load i32, ptr %2, align 4
  %11 = load i32, ptr %3, align 4
  %12 = add nsw i32 %10, %11
  store i32 %12, ptr %5, align 4
  %13 = load i32, ptr %2, align 4
  %14 = load i32, ptr %3, align 4
  %15 = mul nsw i32 %13, %14
  %16 = call i32 (ptr, ...) @printf(ptr noundef @.str, i32 noundef %15)
  %17 = load i32, ptr %4, align 4
  %18 = add nsw i32 %17, 1
  store i32 %18, ptr %4, align 4
  br label %6, !llvm.loop !6

19:                                               ; preds = %6
  ret i32 0
}
```

Figure 8: IR file for unoptimized loop invariant code motion source code file

File: loop_invariant_code_motion_applied_IR.ll

```
; ModuleID = 'loop_invariant_code_motion_IR.ll'
source_filename = "loop_invariant_code_motion.c"
target datalayout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-i128:128-f80:128-n8:16:32:64-S128"
target triple = "x86_64-unknown-linux-gnu"

@.str = private unnamed_addr constant [20 x i8] c"Value of a*b is: %d\00", align 1

; Function Attrs: noinline nounwind uwtable
define dso_local i32 @main() #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  %5 = alloca i32, align 4
  store i32 0, ptr %1, align 4
  store i32 10, ptr %2, align 4
  store i32 20, ptr %3, align 4
  store i32 0, ptr %4, align 4
  %6 = load i32, ptr %2, align 4
  %7 = load i32, ptr %3, align 4
  %8 = add nsw i32 %6, %7
  %9 = load i32, ptr %2, align 4
  %10 = load i32, ptr %3, align 4
  %11 = mul nsw i32 %9, %10
  %.promoted = load i32, ptr %4, align 4
  %.promoted1 = load i32, ptr %5, align 1
  br label %12

12:                                               ; preds = %16, %0
  %13 = phi i32 [ %8, %16 ], [ %.promoted1, %0 ]
  %14 = phi i32 [ %18, %16 ], [ %.promoted, %0 ]
  %15 = icmp slt i32 %14, 5
  br i1 %15, label %16, label %19

16:                                               ; preds = %12
  %17 = call i32 (ptr, ...) @printf(ptr noundef @.str, i32 noundef %11)
  %18 = add nsw i32 %14, 1
  br label %12, !llvm.loop !6

19:                                               ; preds = %12
  %.lcssa2 = phi i32 [ %13, %12 ]
  %.lcssa = phi i32 [ %14, %12 ]
  store i32 %.lcssa, ptr %4, align 4
  store i32 %.lcssa2, ptr %5, align 1
  ret i32 0
}
```

Figure 9: IR file for optimized loop invariant code motion source code file

When we compare the transformation passes dead store elimination, function inlining, and loop invariant code motion; an inline function is one for which compiler copies code from the function definition directly into code of the calling function rather than creating a seperate set of instructions in memory. This elminates call-linkage overhead. Dead store elimination eliminates stores when the value stored is never referenced again. For example, if store in certain function does not have corresponding load in the same function then it is unnecessary and is removed. Loop invariant code motion performs the calculation outside the loop and results used within the loop if variables used in computation within a loop are not altered within the loop [4].

References

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

[2] [Online]. Available: https://llvm.org/docs/Passes.html#adce-aggressive-dead-code-elimination.

[3] Jun. 2018. [Online]. Available: https://www.ibm.com/support/pages/what-does-it-mean-inline-function-and-how-does-it-affect-program.

[4] [Online]. Available: https://www.ibm.com/docs/es/aix/7.2?topic=techniques-compiling-optimization.