# Assignment 3 Loop Invariant Code Motion

Due Date: **Apr. 12th**, Total Marks: 100 pts

CSCD70 Compiler Optimizations
Department of Computer Science
University of Toronto

## ABSTRACT

In this assignment, you will implement the Loop Invariant Code Motion (LICM) discussed in class. You will also develop a register allocator according to the interference graph algorithm as is discussed in class. The register allocator will be used for lowering the optimized IR to machine code and have it run on real hardware.

## 1 POLICY

### 1.1 Collaboration

You will work in groups of **two** for the assignments in this course. Please turn in a single submission per group.

### 1.2 Submission

Please submit all your files to MarkUs/GitHub Classroom. Make sure that your submission includes the followings:

```
writeup.pdf

lib/LICM.cpp
lib/RegAllocIntfGraph.cpp
CMakeLists.txt
test/...
```

- A report named `writeup.pdf` that describes the implementation of your passes, and has answers to the theoretical questions.
- Well-commented source code for your *LICM* pass and *IntfGraph* register allocator, together with a build file `CMakeLists.txt` (please write your `CMakeLists.txt` in such a way that all passes can be built, integrated, and tested using the command `cmake .. && make && make test`).
- A subfolder named *tests* that contains all the microbenchmarks used for verification of your code.

## 2 PROBLEM STATEMENT

### 2.1 Loop Invariant Code Motion [40 pts]

In this pass, you will decrease the number of dynamic instructions executed during a loop by identifying and hoisting/sinking out those that are *loop-invariant*, as was discussed in class. Please call your pass *loop-invariant-code-motion* (please note that naming it to *licm* will cause a naming conflict).

In addition to the usual preprocessing *mem2reg*, you should also optimize your code using the LLVM built-in pass loop-simplify to insert loop preheaders where appropriate. If this built-in pass is unable to insert a preheader, you can ignore the loop.

It is recommended to derive from LLVM's `LoopPass`. *You are not allowed to use methods from LLVM that are related to loop-invariance*, which includes but not limited to `isLoopInvariant`, `hasLoopInvariantOperands`, `makeLoopInvariant`.

For each loop, compute the set of loop-invariant instructions. When checking for loop-invariance, you should also include the following additional conditions in Listing 1 for determining whether an instruction is invariant. Hoist to the preheader or sink to the exit all loop-invariant instructions that are candidates for code motion, ensuring that dependencies are preserved.

```
bool isInvariant(Instruction *const I) {
  bool IsInvariant =
      // your implementation goes here

    return isSafeToSpeculativelyExecute(I)
        && !I->mayReadFromMemory()
        && !isa<LandingPadInst>(I)
        && IsInvariant;
}
```

**Listing 1: `isInvariant` Code Snippet**

In your documentation, please make sure to answer the following questions: (1) How you check for loop invariance, and also, why you think we need the additional conditions in Listing 1 (especially the first two)? (2) How you hoist the code that is loop-invariant to the loop preheader, or sink to the loop exit? (3) Which one(s) of the loop structures (`for`, `while`, `do-while`) can LICM work naturally (i.e., without modifying the CFG) and why? What transformations do you need to make for those loop structures which LICM is not directly applicable?

### 2.2 Register Allocation [40 pts]

Most of the optimizations that we have implemented so far are on the IR level. In this part of the assignment, we will switch gear to approach a backend optimization problem. Specifically, you will implement a register allocator using the interference graph algorithm that we have described in class. The register allocator will be used to map the IR to backend-dependent machine assembly code, which can then be compiled and executed on real hardware.

Please make sure that you implement and test your register allocator on a x86_64 machine (which is the architecture of the MathLab machine). The correctness of your register allocator will be verified on the example shown during the register allocation tutorial and the optimized IR from your LICM optimization pass.

# 3 THEORETICAL QUESTIONS

## 3.1 Pointer Analysis [10 pts]

Recall that pointer analysis can be flow insensitive, flow sensitive, or path sensitive. For the code below, show the result of performing each type of pointer analysis for the dereference of pointer p at the last line.

```
S1:  p = malloc();
S2:  q = malloc();
S3:  p = q;
S4:  r = malloc()
S5:  q = r;
S6:  if (a)
S7:      p = q;
S8:  if (b)
S9:      p = r;
S10: if (!a)
S11:     p = malloc();
S12: ? = *p;
```

Recall that pointer analysis can also be context sensitive or insensitive. For the code below, show the result of performing each type of pointer analysis for the dereference of p.

```
int a, b, c, *p, *q;
int main()
{
S1:      f1();
S2:      p = &a;
S3:      f2();
S4:      p = q;
S5:      f3();
}
void f1()
{
S6:      p = &a;
S7:      q = &c;
S8:      f2();
S9:      f3();
}
void f2()
{
S10:     p = &b;
S11:     q = &a;
S12:     f3();
}
void f3()
{
S13:     ? = *p;
}
```

## 3.2 Register Allocation [10 pts]

Suppose that you have a processor with four registers. Consider the following code, where only definitions and uses of interest are shown. Perform the *register allocation* algorithm described in class, showing the following steps for full marks:

(1) live variables
(2) reaching definitions
(3) live ranges
(4) interference graph
(5) final colored graph

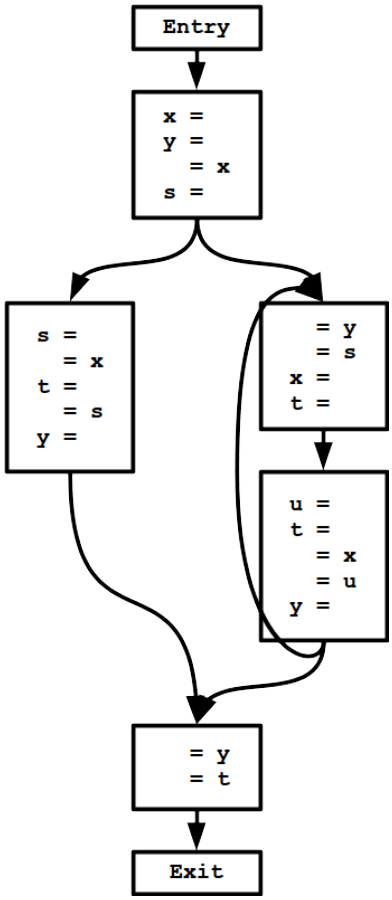


**Figure 1: Code for Register Allocation Analysis**

# 4 FAQ

*Given below is the questions asked during previous offering of the class. If you do not think they fully answer your question, please open a new thread on Piazza.*

**Q: For the interference graph in the theoretical question, should we split the live ranges making $x_{1,2,3}$ etc. or should we treat each variable as a node?**

**A:** In the programming portion, please follow the live intervals given to you by the *LiveIntervals* pass. In the theoretical questions, please treat each variable as a node.