COMP 412, Fall 2023
Lab 2: Local Register Allocation

Table of Contents		
1.	Introduction	1
2.	Overview of the Problem	2
3.	Code Specifications	3
5.	Submitting Your Code	5
6.	Grading Rubric & Honor Policy	6
7.	Honor Policy	7
A.	ILOC Simulator & Subset	8
B.	Tools	10
C.	Timing Results from Prior Years	11
D.	Checklist for Lab 2	13

Critical Dates for the Project	
Code Due Date	10/18/2023
Code Check #1 Due	09/29/2023
Code Check #2 Due	10/11/2023

Please report suspected typographical errors to the instructors via a message on the class Piazza site.

1. Introduction

In this programming assignment you will build a local register allocator—that is, a program that reads in a single block of ILOC code, transforms that block so that it uses a specified number of registers, and writes the transformed block out to the standard output stream. The input and output blocks are both written in the ILOC subset used in Lab 1. Section A.1 of this document describes the ILOC subset and its simulator.

> *For the purposes of this lab, an input program and an output program are considered* ***equivalent*** *if and only if they print the same values in the same order to* stdout *and each data-memory location defined by an execution of the input program receives the same value when the output program executes.*
>
> *The output program may define additional memory locations that are not defined by the input program. In particular, it will define and use new locations to hold values that it decides to spill from memory to registers.*

The input blocks receive input in one of two ways. Either it is hard-coded into the block, or you provide that input on the command line using the simulator's **-i** command-line option.

The concepts and algorithms behind register allocation are explained in two distinct places. First, there are a series of short videos available on the course Canvas Site. You should watch each of these videos. Second, an excerpt from the third edition of the textbook is available on the Canvas Site; this portion of Chapter 13 describes the algorithms that pertain to Lab 2. You should read the Chapter 13 excerpt posted on Canvas.

This document describes the specifications for your program, the policies for the lab, the various logistics associated with the lab, the process for submitting the lab, and the grading

rubric. The document is long, but you should read it to ensure that you understand what is expected of your lab and what tools are available to help.

2. Overview of the Problem

A *register allocator* is a compiler pass that transforms a version of the program that uses an arbitrary number of registers into a version that uses a specified number of registers. That is, the input program might be written to use seventy-three registers and the output program can only use the number of registers available for application use on the target machine.

The number of registers available for the output program is a parameter, called *k*, that is passed into the allocator on the command line. When invoked with a command line, such as

./**412alloc 17 ~comp412/students/ILOC/SLOCs/T001k.1**

The allocator should read the file, **~comp412/students/ILOC/SLOCs/T001k.i**, determine if it is a valid ILOC block, and produce an equivalent block that uses *at most* 17 registers.

To transform the program, the register allocator reads through the operations in the block and makes, at each operation, a decision as to which values should remain in registers and which values should be relegated to storage in memory. It inserts **stores** (*spills*) and **loads** (*restores*) as necessary to ensure that the values are in the right places. It rewrites the code to use the names of the actual hardware registers (r0 through r16 for the example command line).

Your allocator may not apply optimizations other than register allocation. All the **loads**, **stores**, and arithmetic operations that appear in the input program must appear in the output program, in the same relative order.[1] The output block must perform the original computation; it cannot pre-compute the results. *Allocators that perform optimizations other than register allocation will lose substantial credit*. Of course, the output block may use different register names than the input block.

Your allocator **may** remove nops; they have no effect on the equivalence of the input and output of programs. If your allocator performs *rematerialization* (see the Lab 2 Performance Lecture), then it may move **loadI** operations around in ways that cannot happen with a load or a store. Finally, your allocator must not perform optimizations based on the input constants specified in ILOC test block comments; the autograder may use different input values to test your code.

To evaluate your submission, we will use an autograder — a fairly-straightforward python program that will unpack your submission, perform any actions needed to build the executable, and test your executable against a series of test blocks. We will provide a stripped-down version of the autograder so that you can verify that your submission works with the autograder. The "stripped-down" version will not include all of the test files used to assess your lab for a grade.

[1] We say "relative order" because the allocator may insert spills and/or restores between any two operations in the original program.

3. Code Specifications

Your allocator must adhere to the following specifications.

- **Name:** The executable version of your allocator must be named **412alloc.**
- **Behavior:** Your allocator must work in following three modes:

  `412alloc -h` When passed the -h flag, 412alloc must print a list of the valid command-line arguments that it accepts, along with a concise explanation for that option. That list should include the arguments described in this table, as well as any others that your allocator supports. After printing the message, 412alloc should sto.

  `412alloc -x <name>` The -x flag will only be used for Code Check 1. Again, <name> is a Linux pathname. With this flag, 412alloc should scan and parse the input block. It should then perform renaming on the code in the input block and print the results to the standard output stream (stdout). **lab2_ref** does not implement the -x flag.

  `412alloc k <name>` In this format, k is the number of registers available to the allocator ($3 \le k \le 64$) and <name> is a Linux pathname to the file containing the input block. The pathname can be either a relative pathname or an absolute pathname.

  If k is outside the valid range or it cannot open the file specified by <name>, 412alloc should print a reasonable error message and exit cleanly (e.g., no backtrace).

  If the parameters are valid, 412alloc should scan parse, and allocate the code in the input block so that it uses only registers r0 to rk-1 and print the resulting code to the standard output stream (stdout).

  In each mode, **412alloc** should check the input parameters and report any problems on the standard error stream (stderr). All error messages should be printed to the standard error stream (stderr). Normal output should be printed to the standard output stream (stdout).

- **Input File:** The input file will consist of a sequence of ILOC operations (a block) in the same subset described in § A. If **412alloc** cannot read the input file, or the code in the file is not valid ILOC, it should write an error message to the standard error file (stderr). **412alloc** should detect as many errors in the file as it can before quitting.

  If the ILOC code in the input block uses a value from a register that has no prior definition, your allocator should handle the situation gracefully.

  Scanning and *Parsing*: Your register allocator should use your front end from Lab 1 to read and parse the input file. The front end should be prepared to handle large input files. Several large files are available for testing, such as T128k.i from Lab 1.

  Makefile & Shell Script: As in Lab 1, you will submit a tar archive file that contains the source code for your allocator, along with any scripts or **Makefile**s required to create and execute the final code. .[2] For a Java lab, you must submit the code, not a **jar** file.

[2] If you prefer to use another build manager (available on CLEAR), invoke that build manager in your makefile. Your build manager should not leave behind files — particularly hidden files in comp412's home directory.

Lab 2 submissions written in languages that require a compilation step, such as C, C++, or Java, must include a **Makefile**. The autograder will invoke **make** with the two commands:

```
make clean
make build
```

in that order. The **clean** target should remove any files from old attempts to build the executable. The **build** target should ensure that the 412alloc executable is ready to run. If your submission is written in a language that does not require compilation, such as python, then it does not need a **Makefile**.

If your submission does not create a standalone executable named **412alloc**, then it should include a shell script named **412alloc** that accepts the required command-line arguments and passes them to the program. For example, a project written in python[3] named lab2.py could provide an shell script named 412alloc that includes the following instructions:

```
#!/bin/bash
python lab2.py $@
```

To invoke python2, you would use the command "python2" rather than "python".

Similarly, a project written in Java with a jar file named lab2.jar or a class named lab2.class that contains the main function could provide, respectively, one of the following two executable shell scripts, naming it **412alloc.**

```
#!/bin/bash
java -jar lab2.jar $@
```

```
#!/bin/bash
java lab2 $@
```

To ensure that your **412alloc** shell script is executable on a Linux system, execute the following command in the CLEAR directory where your 412alloc shell script resides:

```
chmod a+x 412alloc
```

To avoid problems related to the translation of carriage return and line feed between Windows and Linux, we strongly recommend that you write your shell script and **Makefile** on CLEAR rather than on a Windows system (see also **man dos2unix**).

- **README:** Your submission must include a **README** file (name in uppercase letters) that provides directions for building and invoking your allocator. Include a description of all command-line arguments required for Lab 2 as well as any additional command-line arguments that your allocator supports. The autograder expects that the first two lines of your README file are in the following format:

  ```
  //NAME: <your name>
  //NETID: <your netid>
  ```

  Note that there are no spaces after the slashes, and that the keywords **NAME** and **NETID** are in all capital letters. Your name should be capitalized normally. Your netid should be lowercase characters and numbers.

[3] Alternatively, you can write a python program that runs directly. As an example, see the lab 1 testing script on CLEAR. Its first line is a comment that tells the shell to invoke python; the main routine parses argv and argc.

- **Programming Language:** You may use any programming language provided on Rice's CLEAR facility, except for Perl. Your goal should be to use a language that is available on CLEAR, in which you are comfortable programming, for which you have decent debugging tools, and that allows you to easily reuse code in Lab 3. When coding, be sure to target the version of your chosen programming language that is available on CLEAR**.**
- **USE CLEAR:** Your submission must work on CLEAR. If the code must be translated (e.g., compiled, linked, turned into a jar or an a.out), that must happen on CLEAR. Test the code on CLEAR; test it in the scaled-down autograder. Students have, in previous years, found differences between language and library versions on their laptops and on CLEAR.

4. Code Checks

Lab 2 has two intermediate code checks. We will post directions on how to demonstrate that your code passes the code checks. The code checks are intended to keep your progress on track, time wise. You get full credit at the due date, half credit two days late, and no credit beyond two days late. Late days do not apply. (The grading rubric is discussed in § 5.)

Code Check #1: The due date for code check #1 is shown on page 1. To pass code check #1, **412alloc** must correctly scan, parse, perform register renaming, and print the resulting renamed ILOC block to stdout. The code check #1 script and test blocks are located on CLEAR in the directory ~**comp412/students/lab2/code_check_1/**.

Code Check #2: The due date for code check #2 is shown on page 1. To pass code check #2, **412alloc** must correctly perform register allocation on a limited set of test blocks. (That is, the allocated code must produce the correct answers when run with only *k* registers on the ILOC simulator.) The code check #2 script and the test blocks are found in **~comp412/students/lab2/code_check_2/**.

The code check directories on CLEAR contain a **README** file that describes how to invoke the code check script and interpret the results. As before, you may need an executable shell script that to conform to the interface. A working **Makefil**e is not critical for the code checks, but we recommend that you create your **Makefile** or script before the first code check and use it while developing your code.

5. Submitting Your Final Code

Due date: The due date for your code submission is shown on page 1. All work is due at 11:59PM on the specified day. Individual extensions to this deadline will not be granted.

Early-Submission Bonus: Submissions received before the due date will be awarded an additional two points per day up to a maximum of four points.

Late Penalty: Submissions received after the due date will lose two points per day. Late code will be automatically accepted for seven days after the due date. To submit code after that deadline, you must obtain express permission from the instructors. Contact the instructors via *a private note on Piazza* to request permission.

Late Penalty Waivers: To cover situations that inevitably arise, we will waive up to six days of late penalties per semester when computing your final COMP 412 grade. Note that we choose where those days apply, after all grades are complete. We will apply these "grace" days in a way that maximizes benefit to you.

Submission Details: To submit your work, you must create a tar archive that contains your submission, including (1) the source code for the register allocator, (2) the **Makefile** and/or shell script, (3) the **README** file, and (4) any other files that are needed for the autograder to build and test your code. The tar file should unpack into the current working directory. The **README**, **Makefile,** and executable script, if any, must reside in that top-level directory.

The document "**NoteOnTarFiles.pdf**", available on both the COMP 412 Canvas Site and in the **~comp412/students/lab1** directory on CLEAR, provides more details about the structure of the submission file system and the tar archive. You can test your archive using the scaled-down autograder. If your archive does not work with the autograder, that will reduce the points you receive in the conforms-to-specs part of the grading rubric.

If your allocator does not work, include in your tar file a file named STATUS that contains a brief description of its current state. You may include ILOC files that your lab handles correctly and ILOC files that it handles incorrectly. List those file names in the STATUS file.

Name the tar file with your Rice netid (e.g., if your netid is jed12, you would name the archive jed12.tar). To submit your tar file, move it to CLEAR and execute the command

~comp412/bin/submit_2 ***<tar file name>***

The submit_2 script will create a copy of your tar archive, timestamp it, and send an email confirmation to you and to the comp412 email account (for record-keeping). You should keep the tar archive until the end of the semester to record what you submitted.

6. Grading Rubric

The Lab 2 grade accounts for 22% of your final COMP 412 grade. The code rubric is based on 100 points, allocated as follows. Correctness and cycle count will be determined using the Lab 2 ILOC simulator.

- **10 points** for passing code check #1 by its due date (see table on page 1). 5 points will be awarded for passing code check #1 within three days of the due date. No points for code check #1 will be awarded after that time. Grace days do not apply to this deadline.
- **10 points** for passing code check #2 by its due date (see table on page 1). 5 points will be awarded for passing code check #2 within three days of the due date. No points for code check #2 will be awarded after that time. Grace days do not apply do this deadline.
- **10 points** for adherence to the Lab 2 code specifications and submission requirements.
- **30 points** for correctness of the allocated code produced by the final submission. Here, correctness means that running the allocated code produces the correct answer.
- **20 points** for effectiveness, measured as the number of cycles required to run the allocated code produced by your final submission on the Lab 2 ILOC simulator. An

allocator that gets within 10% of the cycle counts for lab2_ref will receive full credit for effectiveness. You can use the spreadsheet **Lab2SpreadSheet.xlsx**, in the lab 2 documents folder, to estimate effectiveness on the **report blocks**. The **report blocks** are a subset of the blocks used to grade your allocator.

- **20 points** for scalability and efficiency. These points will be awarded for two different criteria: efficiency and scalability. The autograder will time your allocator on **SLOCs** timing blocks, found at **~comp412/students/ILOC/Scalability/SLOCs**. Points toward your grade will be based on the measured behavior of your submission.

  Scalability: The register allocator should display linear scaling — that is, a doubling of the input size should produce growth of no more than 2x in the runtime. You can see this by plotting runtime as a function of non-comment lines in the input file; do not use a logarithmic scale on either axis

  Efficiency: The table below shows the measured runtime on block T128k.i required for full credit and the time that will produce zero credit. Runtimes between full credit and zero credit will receive partial credit, on a linear scale between the breakpoints. The specific breakpoints were determined by analyzing the lab 2 submissions from the Fall 2020 class.

Language	Full Credit	No Credit
C	$\leq$ 1.0 second	$\geq$ 2.0 seconds
C++	$\leq$ 2.0 second	$\geq$ 4.0 seconds
Java	$\leq$ 2.75 seconds	$\geq$ 10.0 seconds
Python	$\leq$ 8.0 seconds	$\geq$ 20.0 seconds
Go	$\leq$ 1.0 second	$\geq$ 2.0 seconds

  For languages not shown in the table, the instructor will determine a set of breakpoints based on the language, its implementation, and, perhaps, some additional testing.

7. Honor Policy

Your submitted allocator source code and **README** file must consist of code and/or text that you wrote, not edited or copied versions of code and/or text written by others or in collaboration with others. You may not look at COMP 412 code from past semesters. Your submitted code may not invoke the COMP 412 reference allocator, or any other allocator that you did not write.

You are free to collaborate with current COMP 412 students when preparing your **Makefile** and/or shell script and to submit the results of your collaborative **Makefile** and/or shell script efforts. However, as indicated in the previous paragraph, all other Lab 2 code and text submitted must be your own work, not the result of a collaborative effort.

You are welcome to discuss Lab 2 with the COMP 412 staff and with students currently taking COMP 412. You are also encouraged to use the archive of test blocks produced by students in previous semesters. However, we ask that you not make your COMP 412 labs available to students (other than COMP 412 TAs) in any form during or after this semester. We ask that you not place your code anywhere on the Internet that is viewable by others.

Appendix A. ILOC Simulator & Subset

ILOC Simulator: An ILOC simulator, its source, and documentation are available in the subtree under **~comp412/students/lab2** on CLEAR. The source code and documentation are available. If you need to build a private copy of the simulator, § 7.1 of the simulator documentation explains the various configuration options.

The simulator builds and executes on CLEAR. You can either run the simulator from **comp412's** directory or copy it into your local directory. You can build a copy to run on your laptop. It appears to work on other OS implementations but that is not guaranteed. Your allocator will be tested and graded on CLEAR, so you should be sure to test it on CLEAR.

ILOC Subset: Lab 2 input and output files consist of a single *basic block*[4] of code written in a subset of ILOC, detailed in the following table. ILOC is case-sensitive.

Syntax				Meaning	Latency
load	r1	=>	r2	r2 ← MEM(r1)	3
loadI	x	=>	r2	r2 ← x	1
store	r1	=>	r2	MEM(r2) ← r1	3
add	r1, r2	=>	r3	r3 ← r1 + r2	1
sub	r1, r2	=>	r3	r3 ← r1 - r2	1
mult	r1, r2	=>	r3	r3 ← r1 * r2	1
lshift	r1, r2	=>	r3	r3 ← r1 << r2	1
rshift	r1, r2	=>	r3	r3 ← r1 >> r2	1
output	x			prints MEM(x) to stdout	1
nop				idle for one cycle	1

All register names have an initial lowercase r followed immediately by a non-negative integer. Leading zeros in the register name are not significant; thus, r017 and r17 refer to the same register. Arguments that do not begin with r, which appear as x in the table above, are assumed to be non-negative integer constants in the range 0 to $2^{31} - 1$. The assignment arrow is composed of an equal sign followed by a greater than symbol, as shown (=>).

[4] A *basic block* is a maximal length sequence of straight-line (*i.e.*, branch-free) code. We use the terms *block* and *basic block* interchangeably when the meaning is clear.

Each ILOC operation in an input block must begin on a new line and be completely contained on that line.[5] Whitespace is defined to be any combination of blanks and tabs. ILOC opcodes must be followed by whitespace. Whitespace preceding and following all other symbols is optional. Whitespace is not allowed within operation names, register names, or the assignment arrow. A double slash (//) indicates that the rest of the line is a comment and can be discarded. Empty lines and nops in input files may also be discarded.

Simulator Usage Example: If test1.i is in your present working directory, you can invoke the simulator on test1.i in the following manner to test your register allocator:

/clear/courses/comp412/students/lab2/sim -r 5 -i 2048 1 2 3 < test1.i

This command will cause the simulator to execute the instructions in test1.i, print the values corresponding to ILOC output instructions, and display the total number of cycles, operations, and instructions executed.

The -r parameter is optional and restricts the number of registers the simulator will use. (In the example, the simulator only uses 5 registers, named r0, r1, r2, r3, and r4.). You can use the –r parameter to verify that code generated by your allocator uses at most *k* registers. You should not use -r when running the original (non-transformed) report and timing blocks.

The -i parameter is used to fill memory, starting at the memory location indicated by the first argument that appears after -i, with the initial values listed after the memory location. The Lab 2 ILOC simulator has byte addressable memory, but the ILOC subset for Lab 2 and Lab 3 only allows word-aligned accesses. So, in the above example, 1 will be written to memory location 2048, 2 to location 2052, and 3 to location 2056. (This means that, before the memory locations are overwritten during program execution, "output 2048" will cause 1 to be printed by the simulator, "output 2052" will cause 2 to be printed, etc.) When computing addresses for new spill locations, your allocator must generate word aligned addresses (e.g., *addr* MOD 4 = 0).

The -x parameter will be used to verify that your allocator passes the Lab 2 code check. For example, if your 412alloc implementation produces a file of renamed ILOC code called renamed_block.i, the following command can be used to check whether renaming was correctly performed:

/clear/courses/comp412/students/lab2/sim -x < renamed_block.i

See the ILOC simulator document for additional information about supported command-line options. (Note that the command-line options -d, -s, and -c are not relevant for Lab 2.)

ILOC Input Blocks: A large collection of ILOC input blocks is available on CLEAR, in the directory **~comp412/students/ILOC/**. Each block has the //SIM INPUT: and //OUTPUT: specifications to allow the various testing scripts to check for correctness. If you experience problems with the input blocks, please submit bug reports to the course Piazza site.

The Lab 2 **report blocks** are available on CLEAR in **~comp412/students/lab2/report/**. The timing blocks used to produce timing information for Lab 2 are available on clear in **~comp412/students/ILOC/Scalability/SLOCs/**.

[5] Both Carriage returns (CR, \r, 0x0D) and line feeds (LF, \n, 0x0A) may appear as in end-of-line sequences.

B. Tools

B.1 Reference Allocator

To help you understand the functioning of a local register allocator and to provide an exemplar for your implementation and debugging efforts, we provide the COMP 412 reference allocator. The reference allocator is a C implementation of the allocator. It follows the basic outline of the algorithm presented in class; it pays careful attention to how it generates spill code. You can improve your understanding of register allocation by examining its output on small blocks. The output code includes comments that are intended to help you understand what it did.

You can use the reference allocator to determine how well your allocator performs in terms of effectiveness (quality of the allocation that your allocator generates) and efficiency (runtime of your allocator).

The COMP 412 reference allocator can be invoked on CLEAR as follows:

~comp412/students/lab2/lab2_ref k <name>

where k is an integer (k > 2) that specifies the number of registers available to the allocator and <name> is a valid Linux pathname relative to the current working directory that names an input file. For a description of the complete set of flags supported by the COMP 412 reference allocator, enter the following command on CLEAR:

~comp412/students/lab2/lab2_ref –h

Note that the COMP 412 reference allocator can only be run on CLEAR.

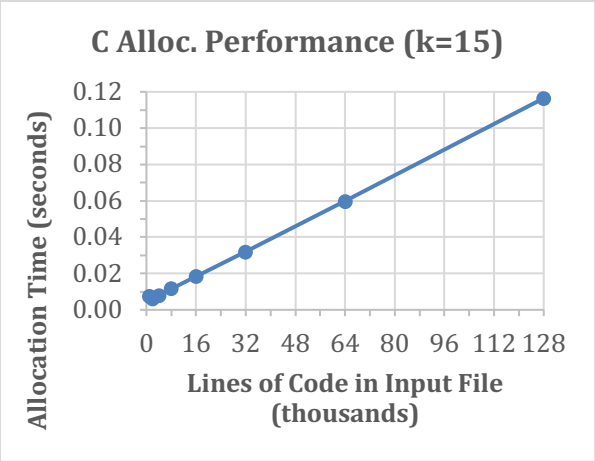
B.2 Testing and Timing Scripts

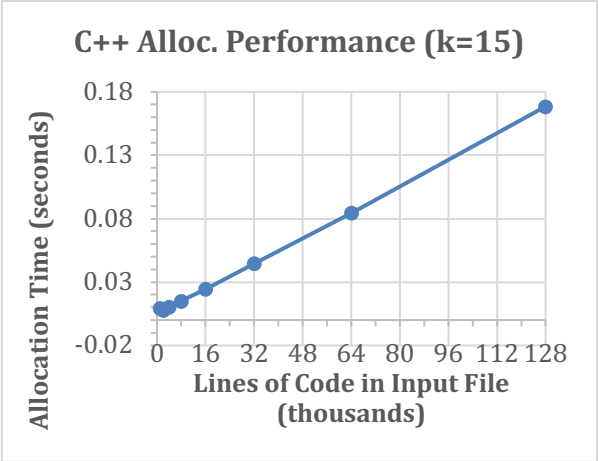
Scripts to test your allocator's correctness and efficiency are available on CLEAR in the directory **~comp412/students/lab2/**. Directions are provided in the related **README** files.

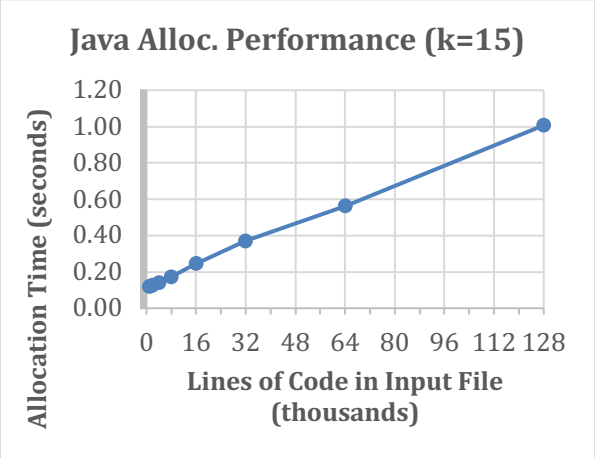
C. Timing Results from Prior Years

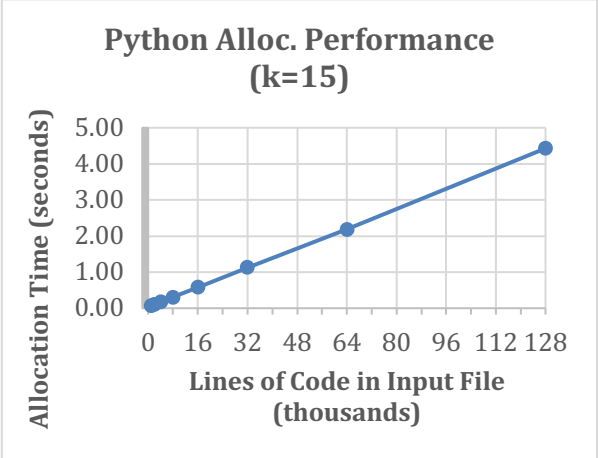
The graphs below show the timing results for the fastest implementation written in each of C, C++, Java, Python, and Go, from Fall 2021. In addition, we include graphs for Ruby (2014), OCaml (2015), Haskell (2015), and R (2014). These graphs should provide you with a reasonable notion of what kind of efficiency a student can achieve in the allocator project.

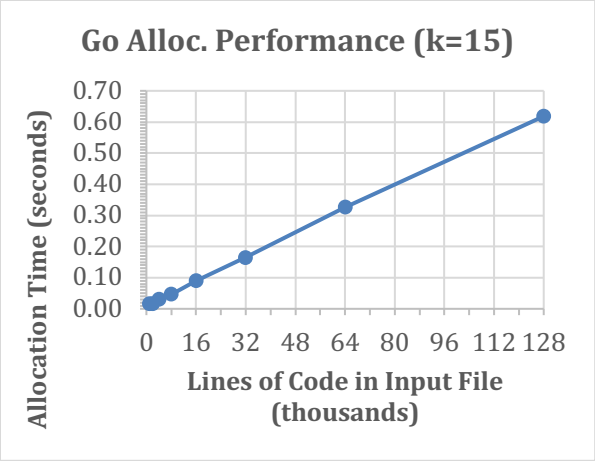
Note the units on the vertical axes of the graphs. They change significantly between languages.

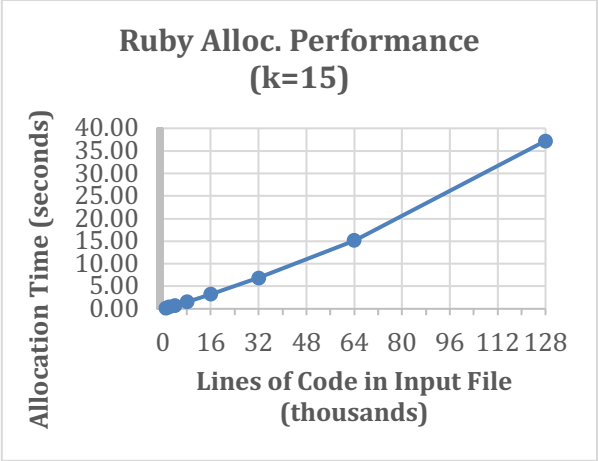


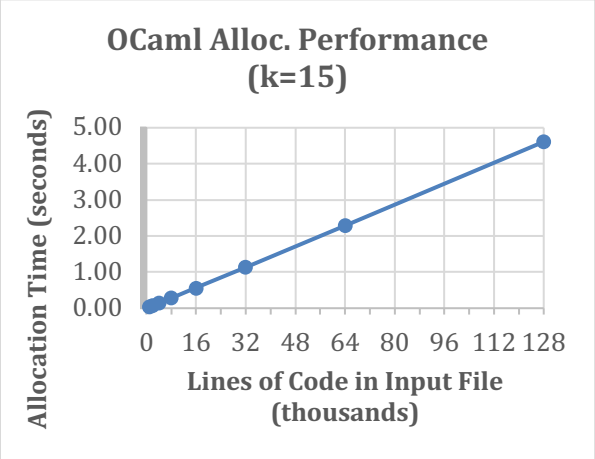


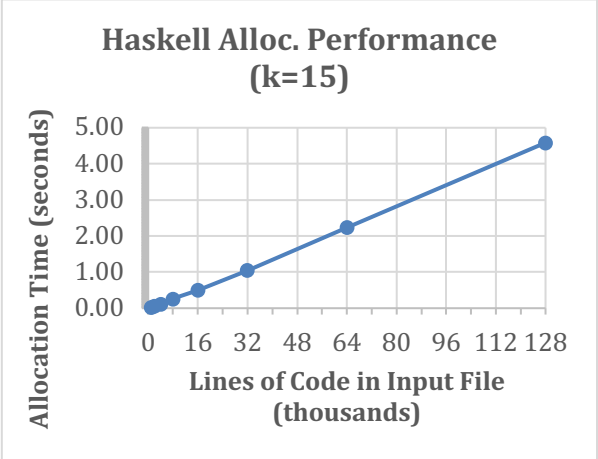


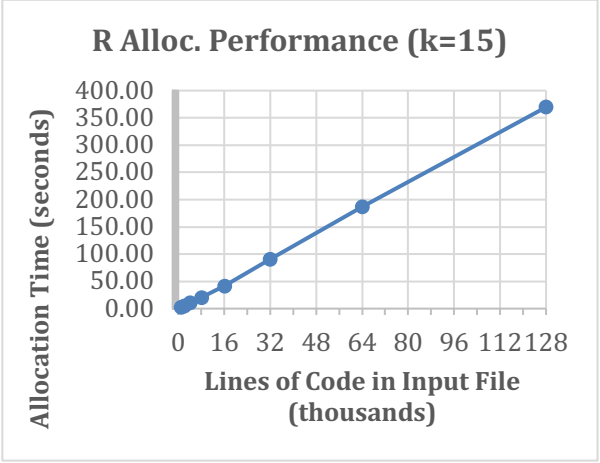



Go Alloc. Performance (k=15)
Allocation Time (seconds)
0.70
0.60
0.50
0.40
0.30
0.20
0.10
0.00
0 16 32 48 64 80 96 112 128
Lines of Code in Input File (thousands)


Ruby Alloc. Performance (k=15)
Allocation Time (seconds)
40.00
35.00
30.00
25.00
20.00
15.00
10.00
5.00
0.00
0 16 32 48 64 80 96 112 128
Lines of Code in Input File (thousands)


OCaml Alloc. Performance (k=15)
Allocation Time (seconds)
5.00
4.00
3.00
2.00
1.00
0.00
0 16 32 48 64 80 96 112 128
Lines of Code in Input File (thousands)


Haskell Alloc. Performance (k=15)
Allocation Time (seconds)
5.00
4.00
3.00
2.00
1.00
0.00
0 16 32 48 64 80 96 112 128
Lines of Code in Input File (thousands)


R Alloc. Performance (k=15)
Allocation Time (seconds)
400.00
350.00
300.00
250.00
200.00
150.00
100.00
50.00
0.00
0 16 32 48 64 80 96 112128
Lines of Code in Input File (thousands)

D. Checklist for Lab 2

The following high-level checklist is provided to help you track your progress on Lab 2.

☐ Implement a bottom-up local register allocator based on the algorithm in § 13.3.2 (pages 686–689) of *Engineering a Compiler (EAC2e)*, and the discussion in class and the tutorial sessions. Submit your allocator by **October 18, 2023**.

☐ Use the ILOC simulator, which is described in § A-1, to ensure that the allocated code produced by your program is correct—that is, it produces an equivalent sequence of operations (see § C-1 for a definition of "equivalent")—and to measure the number of cycles that the ILOC simulator requires to execute each allocated block. A variety of ILOC input blocks are available for you to use when testing your allocator. See § A-2 for details.

☐ Test your allocator thoroughly on CLEAR. It will be graded on CLEAR.

☐ Ensure that your code passes Lab 2 code check #1. To pass the code check, your code must correctly scan, parse, perform register renaming, and print the resulting renamed ILOC block to stdout. See § C-3 for details. To receive full credit, submit results that demonstrate that your allocator passes code check #1 on or before **September 28,2023.**

☐ Ensure that your code passes Lab 2 code check #2. To pass the code check, your code must conform to the user interface described in § C-2 and must perform register allocation correctly and print the resulting allocated ILOC block to stdout. See § C-3 for details. To receive full credit, submit results that demonstrate that your allocator passes code check #2 on or before **October 11, 2023**.

☐ Spend the rest of your time improving the effectiveness of the allocator, as measured by the number of cycles that the simulator reports when it runs the allocated code, and the efficiency of the allocator, as reported by the auto-timer.

To produce an effectiveness grade, we will run the allocator on each of the report blocks and on a set of private blocks. We will test each code with k set to 3, 4, 5, 6, 8, and 10. The number of points awarded to an allocator for effectiveness will be based on a comparison of the cycle counts produced by the allocator and the cycle counts produced by lab2_ref on the same input file and k value.

Change History:
Major rewrite, April 2022
Updated dates in Checklist, September< 2023.
Deleted Report and Test Block, June 2023 and, again, September 2023.