CMR TECHNICAL CAMPUS

UGC AUTONOMOUS

Accredited by NBA & NAAC with 'A' Grade

Approved by AICTE, New Delhi and JNTUH. Hyderabad

Academic Year	:
Name of the Student	:
Roll No	:
Year	: B. Tech I/II/III/IV
Semester	: I/II
Section	:
Branch	:
Name of the Laboratory	:
Batch No.	:
Title of the Lab Report/Project	:
	:
Date	:
Signature of the Student	:

LABORATORY REPORT/PROJECT & PRESENTATION				
Problem Statement & Objectives	Design & Methodology	Implementation & Results	Total Marks	Final Marks
10	15	15	40	10

Remarks/Comments by the Faculty:

Name of the Faculty :

Signature of the Faculty :

ABSTRACT

The aim of this project implements a *region-based register allocator* that efficiently assigns program variables to CPU registers using compiler optimization techniques. The system parses an intermediate representation (IR), builds a control flow graph (CFG), performs live variable analysis, and partitions the program into regions based on strongly connected components. Within each region, an interference graph and graph coloring algorithm are used to minimize register spills. Developed in Python with networkx and matplotlib, the project visualizes allocation decisions and demonstrates practical concepts in compiler design, dataflow analysis, and code optimization.The project serves as a foundation for advanced work in code generation, register allocation strategies, and compiler optimization research.

TABLE OF CONTENTS

1. INTRODUCTION

A register allocator is a key component of a compiler's backend responsible for efficiently mapping program variables to a limited set of CPU registers. Since registers are much faster than memory, effective register allocation significantly improves program performance and reduces execution time. Traditional allocation methods, such as linear scan and global graph coloring, often struggle with scalability or precision when dealing with large, complex control-flow structures.

This project focuses on developing a **Region-Based Register Allocator**, an advanced approach that partitions a program's control flow graph (CFG) into smaller **regions** based on strongly connected components. Each region is then optimized independently, enabling better handling of loops and localized register pressure. The allocator performs **live variable analysis**, builds **interference graphs**, and applies a **graph coloring algorithm** to assign registers efficiently while minimizing variable spilling to memory.

The system is implemented in Python using networkx for graph construction and matplotlib for visualization.

2. LITERATURE SURVEY

In Developing a region-based register allocator requires understanding compiler optimization, dataflow analysis, and graph-based allocation techniques.

1.**Fundamentals of Register Allocation:**

Register allocation maps program variables to limited CPU registers, improving execution speed and reducing memory access.

2.**Existing Allocation Techniques:**

Graph-coloring allocators (Chaitin et al.) provide high accuracy but are complex, while linear scan allocators offer faster but less optimal performance.

3.**Region-Based Approaches:**

Region-based allocation partitions the control flow graph (CFG) into smaller regions for localized optimization, improving scalability and reducing register pressure.

4.**Dataflow and Interference Analysis:**

Live variable analysis and interference graphs help detect conflicts between variables, enabling efficient graph-coloring–based register assignment.

5. **Optimization Techniques:**

Spill cost estimation, live-range splitting, and coalescing further enhance allocation efficiency in modern compilers like LLVM and GCC.

.

3. ANALYSIS AND DESIGN

System Analysis

Functional Requirements:

• IR Parsing: Read and interpret the intermediate representation (IR) code to extract instructions and variables.
• CFG Construction: Build a Control Flow Graph (CFG) representing program execution flow between basic blocks.
• Liveness Analysis: Perform live variable analysis to identify variable lifetimes and usage.
• Region Partitioning: Divide the CFG into strongly connected components (regions) for localized register allocation.
• Register Allocation: Use graph coloring to assign variables to registers and minimize spilling.
• Visualization: Generate interference graphs and register pressure plots for each region.
• Code Generation: Produce allocated IR code showing assigned registers and spill locations.

Non-functional Requirements:

• **Performance:** Ensure efficient allocation with minimal computation time for large programs.
• **Scalability:** Handle complex CFGs by region-based partitioning and modular analysis.
• **Usability:** Provide clear visual outputs and readable allocated code for analysis.
• **Maintainability:** Use modular Python design with separate components for parsing, CFG, and allocation.
• **Reliability:** Accurately compute liveness and register assignments to maintain program correctness.

System Design

Architectural Design

1. Input & Parsing Layer:
Reads the intermediate representation (IR) and converts each instruction into a structured format for analysis.

2. Control Flow Graph (CFG) Layer:
Builds a CFG with basic blocks and directed edges representing program flow using networkx**.**

3. Liveness Analysis Layer:
Performs live variable analysis to determine variable lifetimes and dependencies across blocks**.**

4. Region Partitioning Layer:
Divides the CFG into regions based on strongly connected components (SCCs) for localized optimization.

5. Register Allocation Layer:
Constructs interference graphs and applies graph coloring to assign registers efficiently and minimize spills.

6. Code Generation Layer:
Produces optimized IR code with assigned registers and spill handling.

7. Visualization Layer:
Generates interference graphs and register pressure plots using matplotlib for performance analysis.

Flow Diagram

1. Start Execution:

The system begins by reading the input Intermediate Representation (IR) file and initializing all data structures required for analysis.

2. IR Parsing:

Each line of the IR is parsed to extract instructions, variables, and labels, converting them into structured objects for further processing.

3. CFG Construction:

A Control Flow Graph (CFG) is built by dividing the program into basic blocks and linking them through branches, gotos, and fall-through paths.

4. Liveness Analysis:

The system performs live variable analysis to determine which variables are active at each program point, identifying possible interferences.

5. Region Formation:

The CFG is partitioned into strongly connected components (regions) using graph algorithms to localize optimization and simplify allocation.

6. Register Allocation:

For each region, an interference graph is generated, and a graph coloring algorithm assigns registers while minimizing memory spills.

7. Code Generation and Visualization:

Optimized IR code is produced showing register assignments. Interference graphs and register pressure plots are visualized for performance analysis

4. IMPLEMENTATION

4.1 STEPS TO EXECUTE:

Step 1: Set Up the Environment

1. **Install required Install Required Packages:**

   >>pip install networkx matplotlib

2. Ensure Python 3.8 or above is installed.

   >>>pip install cryptography tkinter websocket-server

3. Create the project folder structure with the main script (region_alloc_cfg.py) and input file (program.ir).

Step 2: Prepare the Input IR File

1. Inside the data or project directory, create a file named program.ir.

2. start:
   i = 0
   sum = 0
   L1:
   temp = i * 1
   if temp goto L2
   goto L3
   L2:
   sum = sum + i
   i = i + 1
   goto L1
   L3:
   print sum

Step 3: Run the Allocator

1. **Open a terminal or command prompt in the project directory.**.
2. **Ensure the following files exist:**

   >>python region_alloc_cfg.py program.ir

3. The system will read the IR, build the control flow graph (CFG), and perform live variable and region-based register allocation.

Step 4: View the Output

1. **The console will display:**
   - Number of basic blocks and CFG edges
   - Liveness sets (IN / OUT variables)
   - Region partitions (based on SCCs)
   - Register allocation summary
   - Optimized allocated code
2. Example: Region 1: i -> r1, sum -> r2, temp -> r3

Step 5: Check Generated Files

The following files will be automatically created in the output directory:

- region_0_graph.png, region_1_graph.png → Interference graphs
- register_pressure.png → Register pressure plot
- allocated_code.txt → Optimized code with register assignments

Open the images to visualize how registers are allocated across regions.

Step 6: Validate Execution

1. Verify that each region's graph shows variables and their interference links.
2. Confirm that register assignments are consistent and no variable conflicts occur.
3. Ensure that the allocated code matches the logic of the input IR (no semantic errors).

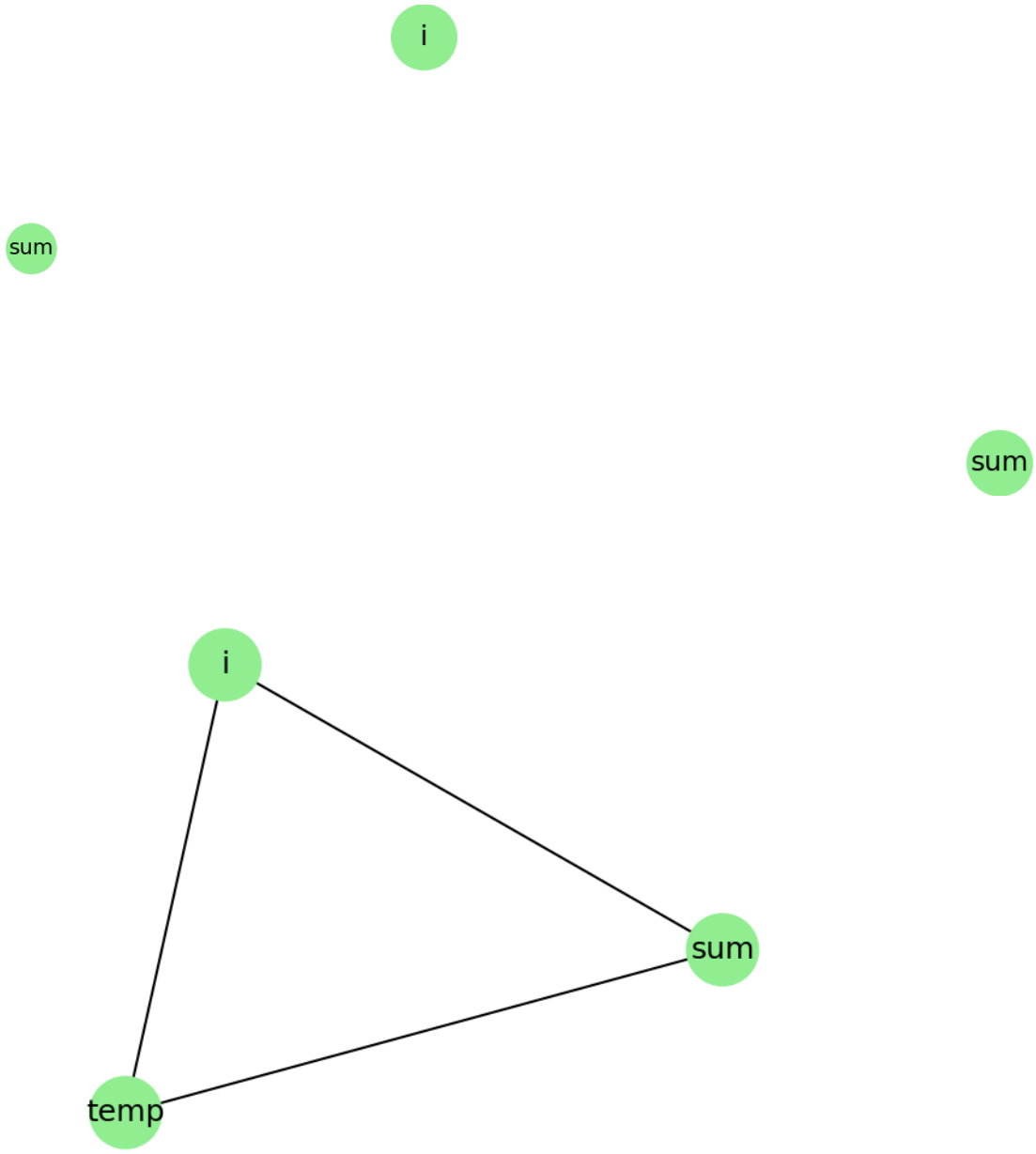
4.2 CODE

Server .py

```
import import asyncio
import json
import ssl
import websockets

# ----- Build CFG: edges between blocks -----
cfg = defaultdict(set)
for bid, blk in enumerate(blocks):
last_idx, last_ins = blk[-1]
if last_ins.type == "goto":
        tgt_idx = label_to_idx[last_ins.label]
cfg[bid].add(idx_to_bid[tgt_idx])
elif last_ins.type == "if":
tgt_idx = label_to_idx[last_ins.label]
cfg[bid].add(idx_to_bid[tgt_idx])
# fall-through if exists
if bid + 1 < len(blocks):
        cfg[bid].add(bid + 1)
 else:
# fall-through if exists
if bid + 1 < len(blocks):
        cfg[bid].add(bid + 1)

# For visualization / sanity
print(f"Blocks formed: {len(blocks)}")
for bid, blk in enumerate(blocks):
        print(f" B{bid}: instr idxs {[i for i,_ in blk]}")
        print("CFG edges:")
        for k, v in cfg.items():
                print(f" B{k} -> {sorted(v)}")
```

```
        # ----- Liveness analysis on CFG (backwards) -----
        def defs_uses_block(blk):
         defs, uses = set(), set()
        for idx, ins in blk:
                    if ins.type == "assign":
                   # dest is a def
        defs.add(ins.dest)
        # sources are uses if they are names (alphabetic)
        if ins.op is None:
                    if isinstance(ins.op1, str) and ins.op1.isalpha():
                            uses.add(ins.op1)
                    else:
                    if isinstance(ins.op1, str) and ins.op1.isalpha():
                            uses.add(ins.op1)
                    if isinstance(ins.op2, str) and ins.op2.isalpha():
                             uses.add(ins.op2)
                    elif ins.type == "if":
                    if isinstance(ins.op1, str) and ins.op1.isalpha():
                            uses.add(ins.op1)
                    elif ins.type == "print":
                    if isinstance(ins.op1, str) and ins.op1.isalpha():
                             uses.add(ins.op1)
                    # labels/goto have no uses/defs
        return defs, uses

        self.title("Advanced Secure Password Manager")
        self.geometry("600x400")
        self.db = load_db()
        self.aes_key = None
        self.priv = None
        self.pub = None

        self.create_login_frame()

def create_login_frame(self):
        self.login_frame = tk.Frame(self)
        self.login_frame.pack(pady=50)

        tk.Label(self.login_frame, text="Master Password:").pack(pady=5)
         self.master_entry = tk.Entry(self.login_frame, show="*")
        self.master_entry.pack(pady=5)
        tk.Button(self.login_frame, text="Login", command=self.login).pack(pady=10)
```

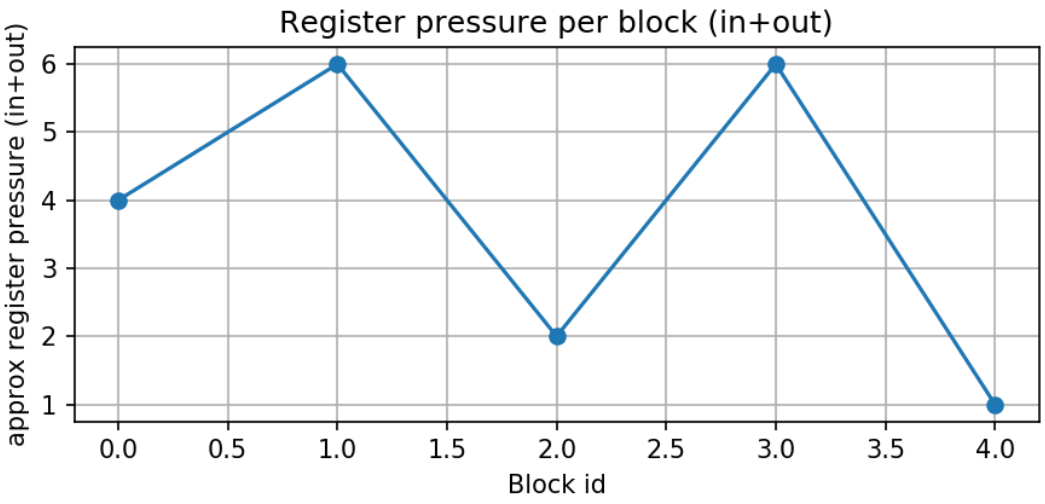
5. TESTING AND DEBUGGING



```
CFG edges:
 B0 -> [1]
 B1 -> [2, 3]
 B2 -> [4]
 B3 -> [1]

Liveness (per block):
 B0: IN=['temp'] OUT=['i', 'sum', 'temp']
 B1: IN=['i', 'sum', 'temp'] OUT=['i', 'sum', 'temp']
 B2: IN=['sum'] OUT=['sum']
 B3: IN=['i', 'sum', 'temp'] OUT=['i', 'sum', 'temp']
 B4: IN=['sum'] OUT=[]
```

```
Globally spilled vars: []

=== Allocated (conceptual) code ===
// Block B0:
start:
    r1 = 0
    r1 = 0
// Block B1:
L1:
    r3 = i * 1
    if temp != 0 goto L2
// Block B2:
    goto L3
// Block B3:
L2:
    r1 = sum + i
    r1 = i + 1
    goto L1
// Block B4:
L3:
    print sum
```

```
Blocks formed: 5
 B0: instr idxs [0, 1, 2]
 B1: instr idxs [3, 4, 5]
 B2: instr idxs [6]
 B3: instr idxs [7, 8, 9, 10]
 B4: instr idxs [11, 12]
```



6. CONCLUSION

In conclusion, this project successfully demonstrates the design and implementation of a Region-Based Register Allocator — an advanced compiler optimization technique that efficiently assigns variables to a limited number of registers using control-flow graph (CFG) analysis, liveness computation, and graph coloring.

By dividing the program into regions (strongly connected components), the allocator performs localized optimization, reducing register conflicts and improving scalability for medium and large programs.
The integration of visualizations — including interference graphs and register pressure plots — provides valuable insight into the allocation process and program behavior.

The system effectively simulates a realistic compiler backend stage and serves as a practical example of how region-based register allocation enhances performance over traditional global allocation methods.

7. REFERENCES

Books

 Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2006). Compilers: Principles, Techniques, and Tools (2nd Edition). Pearson.
- Foundational text covering lexical analysis, parsing, semantic analysis, optimization, and code generation.

 Cooper, K. D., & Torczon, L. (2011). Engineering a Compiler. Morgan Kaufmann.
- Discusses practical compiler engineering techniques, including register allocation and dataflow analysis.

 Muchnick, S. S. (1997). Advanced Compiler Design and Implementation. Morgan Kaufmann.
- In-depth treatment of optimization techniques such as region-based analysis and graph coloring.

 Appel, A. W. (2004). Modern Compiler Implementation in C. Cambridge University Press.
- Explains modern register allocation algorithms and interference graph modeling.

 Allen, F. E., & Kennedy, K. (2002). Optimizing Compilers for Modern Architectures: A Dependence-Based Approach. Morgan Kaufmann.
- Focuses on optimization strategies for large-scale programs and region-based transformations.

Research Papers & Journals

 Chaitin, G. J. (1982). Register Allocation & Spilling via Graph Coloring. Proceedings of the SIGPLAN Symposium on Compiler Construction.
- Introduces the classic graph coloring approach for register allocation.

 Briggs, P., Cooper, K. D., & Torczon, L. (1994). Improvements to Graph Coloring Register Allocation. ACM Transactions on Programming Languages and Systems (TOPLAS).
- Proposes enhancements to spill cost estimation and region-based allocation.

 Chow, F. C., & Hennessy, J. L. (1990). The Priority-Based Register Allocation Technique. ACM Transactions on Programming Languages and Systems (TOPLAS).
- Describes heuristic methods for efficient register assignment.

Websites and Online Resources

- **GeeksforGeeks – Compiler Design Tutorials**
  Explains lexical analysis, syntax parsing, and register allocation concepts.
  📎 https://www.geeksforgeeks.org/compiler-design-tutorials/

- **TutorialsPoint – Compiler Design**
  Provides a step-by-step explanation of dataflow analysis and optimization.
  📎 https://www.tutorialspoint.com/compiler_design/index.htm
- **Wikipedia – Register Allocation**
  Overview of register allocation techniques, spilling, and graph coloring approaches.
  📎 https://en.wikipedia.org/wiki/Register_allocation
- **NetworkX Documentation**
  Reference for creating and analyzing control flow graphs (CFG) and regions.
  📎 https://networkx.org/documentation/stable/
- **Matplotlib Documentation**
  Used for visualization of interference graphs and register pressure plots.
  📎 https://matplotlib.org/
    ○