SIMATS SCHOOL OF ENGINEERING

SAVEETHA INSTITUTE OF MEDICAL AND TECHNICAL SCIENCES

CHENNAI-602105

Memory Management Strategies in Compiler Design

A CAPSTONE PROJECT REPORT

Submitted in the partial fulfillment for the award of the degree of

BACHELOR OF ENGINEERING

INFORMATION TECHNOLOGY

Submitted by

CH.POOJITHA (192211809)

B.POOJA (192211757)

M.DEEPTHI REDDY (192211156)

Y.DIVYA SRI (192210138)

Under the Supervision of

Dr. G. Michael

JUNE 2024

DECLARATION:

We are CH.POOJITHA, B.POOJA, M.DEEPTHI REDDY,Y.DIVYA SRI students of Bachelor of Engineering in Information Technology, Department of Computer Science and Engineering, Saveetha Institute of Medical and Technical Sciences, Saveetha University, Chennai, hereby declare that the work presented in this Capstone Project Work entitled "**Building a Code Generator for Novel Programming Languages**" is the outcome of our own bonafide work and is correct to the best of our knowledge and this work has beenundertaken taking care of Engineering

CH.POOJITHA (192211809)

B.POOJA (192211757)

M.DEEPTHI REDDY (192211156)

Y.DIVYA SRI (192210138)

Date:

Place:

CERTIFICATE

This is to certify that the project entitled **"Building a Code Generator for Novel Programming Languages"** submitted by Poojitha,Pooja , Deepthi reddy, Divya sri has been carried out under our supervision. The project has been submitted as per the requirements in the current semester of B. Tech Information Technology.

Faculty Incharge

Dr. G.Michael

Table of Contents:

Capstone Project

Memory Management Strategies in Compiler Design

Slot: B

Course Code: CSA1475

Course Name: Compiler Design for regular language

Names: 1. CH.POOJITHA 192211809

2 B.POOJA 192211757

3. M.DEEPTHI REDDY

4. Y.DIVYA SRI

1.Preliminary Stage

1.1Assignment Description:

Compilers employ various strategies to manage memory during program execution. Static allocation assigns fixed memory at compile time, offering efficiency but limited flexibility. Stack allocation, using a LIFO approach, is efficient for local variables but has size constraints. Heap allocation provides dynamic memory during runtime, ideal for unknown-sized data structures, but requires manual management to avoid memory leaks. Hybrid approaches combine these strategies for optimal memory usage. Additionally, techniques like garbage collection and register allocation further enhance memory management efficiency. The chosen strategy depends on program needs, balancing efficiency, flexibility, and memory usage.Memory pools involve pre-allocating a fixed amount of memory, improving efficiency for repetitive allocations. Stack allocation follows a last-in, first-out model, suitable for function calls and local variables. Heap allocation, managed by a memory allocator, is dynamic and supports variable-sized data but can lead to memory leaks if not properly managed.

1.2Assignment Work Distribution:

·**Project Scope Definition:** This project will delve into the various memory management techniques employed by compilers to efficiently allocate and manage memory during program execution. We will analyse their suitability for different programming scenarios and identify potential optimizations for specific scenarios.

·**Data Collection and Preparation:**

Identify the data sources: the data sources are taken from databases like ACM digital library, IEEE Xplore, and google scholar for articles on memory management techniques in compiler design.

Develop a data collection plan: the data is collected from various research papers like “compiler decided dynamic memory allocation for scratch pad based embedded system”. We have used software like Mendeley or Zotero to efficiently store, organize, and annotate the collected data.

Cleanse and preprocess the collected data to ensure data quality: the collected data is inspected to ensure its direct relevance to the specific memory management techniques being analysed.

Consistency of the project: Throughout the data collection and preparation process, our sources, search strategies, and data manipulation steps to ensure transparency are analysed. all data collection and analysis adhere to ethical research practices, respecting intellectual property rights and properly citing sources.

·**Exploratory Data Analysis (EDA):**

Conduct exploratory data analysis: Analyse the performance characteristics of different memory management strategies like static, stack, heap based on data collected from research papers or compiler benchmarks.

understand the patterns and trends: we will implement histograms, boxplots, or kernel density plots to visualize the distribution of each variable like execution time, memory usage, cache misses for each memory management strategy.

Visualisation for distribution of execution time across strategies:

```
importmatplotlib.pyplotasplt


# Sample data (replace with your actual data)

execution_times = {

        "static": [10, 12, 15, 18, 20],

        "stack": [8, 10, 11, 13, 14],

        "heap": [15, 18, 20, 22, 25]

}


strategies = list(execution_times.keys())
```

```
plt.figure(figsize=(8, 6))

plt.boxplot(execution_times.values(), labels=strategies, notch=True)

plt.xlabel("Memory Management Strategy")

plt.ylabel("Execution Time (ms)")

plt.title("Distribution of Execution Time Across Strategies")

plt.grid(True)

plt.show()
```

sample implementation of graph:

2. Abstract

Efficient memory management is crucial for program performance and resource utilization. However, selecting the optimal strategy (static, stack, heap, hybrid) remains a challenge due to the diverse characteristics of programs and memory usage patterns.This work proposes a data-driven approach foroptimizing memory management in compiler design**.** We leverage machine learning toautomatically assess program characteristicsand predict the performance impactof different strategies. This enables the recommendation of the optimal strategybased onperformance goals and resource constraints. This data-driven approach holds the potential to revolutionize compiler design by enabling the selection of optimal memory management strategies for various program types, leading to significant advancements in program performance, development efficiency, and resource utilization. it also aims toimprove program performance, enhance development efficiency, and mitigate memory-related issues**.**

Keywords: memory management, compiler design, machine learning, program analysis, performance optimization.

3.Introduction

Memory management is a crucial aspect of compiler design responsible for allocating and deallocating memory efficiently during program execution. Different strategies achieve this based on the data's lifetime, access patterns, and size. Memory management significantly impacts a program's performance, correctness, and resource utilization. Efficient allocation minimizes fragmentation, prevents memory leaks, and optimizes memory access times. Efficient memory management directly impacts a program's performance. Choosing the right strategy can significantly reduce memory access overhead, fragmentation, and cache misses, leading to faster execution speeds. This becomes crucial in performance-critical applications like real-time systems or embedded devices. Memory is a finite and valuable resource, especially in resource-constrained environments. Optimizing memory usage through effective strategies can prevent memory leaks, crashes, and system instability.

4. Problem Statement

Compiler designers face the challenge of selecting the most suitable memory management strategy like static, stack, heap, hybrid for different program types and functionalities. Each strategy offers distinct advantages and disadvantages in terms of efficiency, flexibility, and memory usage. Choosing the optimal memory management strategy for a specific program remains a complex task, often requiring manual analysis and experimentation. This can be time-consuming and error-prone, especially for complex programs with diverse memory requirements.Suboptimal memory management choices can lead to significant performance drawbacks, including increased execution time, excessive memory consumption, and potential memory leaks. This can negatively impact user experience, resource utilization, and overall program reliability. To develop a systematic approach for optimizing memory management strategies in compiler design. This approach should leverage data analysis and machine learning techniques to Automatically assess program characteristics, Predict the performance impact and recommend the optimal strategy.

5. Proposed Design work

5.1 Identify the key components: This design employs feature engineering to capture program characteristics relevant to memory usage patterns. The Random Forest model, trained on historical data, predicts the performance impact of different strategies for a new program. The strategy selection module, informed by predictions and program requirements, recommends the optimal approach. This data-driven system aims to automate memory management selection, improving program performance, compiler efficiency, and memory management effectiveness.

5.2 Functionality:The Feature Engineering Module extracts features from the input program. The extracted features are used to train the Random Forest Model. For a new program, the features are extracted and fed into the trained model. The Random Forest Model predicts the performance impact of each memory management strategy. The Strategy Selection Module analyses the predictions, program requirements, and constraints, recommending the optimal strategy.

5.3 Architectural Design:On working with modular design Each component is independent, facilitating development, testing, and maintenance. Utilization of open- source libraries like scikit-learn for feature engineering and Random Forest implementation. Cloud based training Leverage cloud platforms like Google Collab or Amazon Sage Maker for efficient model training on large datasets and integrated the design as an API within the compiler to enable seamless strategy selection during compilation.

6. UI Design:

TheUI design for thismemory management strategies in compiler design would be to develop a "web-based interface".Accessible from any device with a web browser and it Can be shared easily online. Developers and researchers working with compilers or memory management. These elements facilitate user interaction, data processing, and result presentation. The input panel allows users to provide the code for analysis. The analysis section displays the predicted performance impact of different strategies and visualizes the results for easy comparison. The recommendation section highlights the optimal strategy and provides justification. Additional options allow users to customize the analysis based on their preferences and constraints. This web-based interface offers a user-friendly and accessible way to interact with the memory management optimization tool.

7. code:

```
from flask import Flask, request, jsonify

fromsklearn.ensembleimportRandomForestClassifier


app = Flask(__name__)


# Replace with your feature extraction and prediction logic

def predict_performance(code):

    # Extract features from the code

    features = extract_features(code)



    # Make prediction using the trained model
```

```
        prediction = model.predict([features])[0]

        return prediction


# Load the pre-trained model (replace with your training logic)
model = RandomForestClassifier()
model.load_model("memory_management_model.pkl")


@app.route("/predict", methods=["POST"])
def predict():
        ifrequest.method == "POST":
        code = request.form["code"]
        prediction = predict_performance(code)
        returnjsonify({"prediction": prediction})


if __name__ == "__main__":
        app.run(debug=True)
```

This code initializes a Flask application (app) to handle user requests. predict_performancefunction is placeholder function represents logic for extracting features from the code and making predictions using the trained model. We need to implement this based on chosen feature engineering and machine learning techniques. replace the placeholder code with your model loading logic. You'll need to train and save a model beforehand. /predictroute handles POST requests sent by the frontend containing the source code. the code extracts features, makes predictions using the loaded model, and returns the predicted strategy as a JSON response.

INPUT:

{

```
  "code": "def factorial(n):\n    if n == 0:\n        return 1\n    else:\n        return n * factorial(n - 1)"
}
```

OUT PUT:

```
{
  "prediction": "heap"
}
```

8.Conclusion

This project explored the design of a web-based interface for a memory management optimization tool. The proposed solution utilises feature engineering and a machine learning model to predict the performance impact of different memory management strategies on a given program. The UI facilitates user interaction, allowing them to upload source code, view predicted performance metrics, and receive recommendations for the optimal strategy based on their preferences and constraints.This design offers several potential benefits, including improved developer productivity, informed decision-making regarding memory management, and potentially enhanced program performance. Further research and development could involve refining the feature engineering techniques, exploring alternative machine learning models, and integrating the tool within existing development workflows. By continuously improving and expanding its capabilities, this memory management optimization tool has the potential to become a valuable asset for developers and researchers working on memory-intensive applications.

9. BIBLIOGRAPHY

1. Kandemir, Mahmut, and Alok Choudhary. "Compiler-directed scratch pad memory hierarchy design and management." In Proceedings of the 39th annual Design Automation Conference, pp. 628-633. 2002.

2. Li, Lian, Lin Gao, and Jingling Xue. "Memory coloring: A compiler approach for scratchpad memory management." In 14th International Conference on Parallel Architectures and Compilation Techniques (PACT'05), pp. 329-338. IEEE, 2005.

3. Yang, Yi, Ping Xiang, Jingfei Kong, and Huiyang Zhou. "A GPGPU compiler for memory optimization and parallelism management." ACM Sigplan Notices 45, no. 6 (2010): 86-97.

4. Avissar, Oren, Rajeev Barua, and Dave Stewart. "Heterogeneous memory management for embedded systems." In Proceedings of the 2001 international conference on Compilers, architecture, and synthesis for embedded systems, pp. 34-43. 2001.

5. Gannon, Dennis, William Jalby, and Kyle Gallivan. "Strategies for cache and local memory management by global program transformation." In International Conference on Supercomputing, pp. 229-254. Berlin, Heidelberg: Springer Berlin Heidelberg, 1987.

6. Baradaran, Nastaran, and Pedro C. Diniz. "A compiler approach to managing storage and memory bandwidth in configurable architectures." ACM Transactions on Design Automation of Electronic Systems (TODAES) 13, no. 4 (2008): 1-26.

7. Cooper, Keith D., and Timothy J. Harvey. "Compiler-controlled memory." ACM SIGOPS Operating Systems Review 32, no. 5 (2014): 2-11.

8. Zendra, Olivier. "Memory and compiler optimizations for low-power and-energy." arXiv preprint cs/0610028 (2016).

9. Husmann, Harlan Edward. Compiler memory management and compound function definition for multiprocessors. University of Illinois at Urbana-Champaign, 2018.

10. Li, Lingda, and Barbara Chapman. "Compiler assisted hybrid implicit and explicit GPU memory management under unified address space." In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 1-16. 2019.