Enhancing Macro Processors for Modern System Programming

Gandholi Sarat

March 28, 2025

Problems Overview

1. Optimizing Two-Pass Assemblers for Performance and Memory Efficiency

- Traditional two-pass assemblers scan code multiple times, increasing memory usage and processing time.
- Optimization techniques focus on reducing disk I/O and improving symbol resolution speed.

2. Improving Linker and Loader Strategies for Dynamic Libraries

- Dynamic linking reduces binary size but introduces runtime overhead.
- Research focuses on caching, prefetching, and improving security (e.g., DLL hijacking prevention).

3. Enhancing Macro Processors for Modern System Programming

What is a Macro Processor?

- Automates code substitution during preprocessing.
- Enables reusable and concise code.
- Improves readability and maintainability.

Why are Macro Processors Important?

- Used in assemblers, compilers, and scripting languages.
- Forms the basis of C preprocessor (#define macros).
- Essential for low-level systems programming.

Problem Definition

Challenges in Traditional Macro Processors

- Limited language support.
- Complex build systems.
- Lack of debugging support.
- Performance overhead.
- Limited extensibility.

Research Question:

"How can macro processors be improved to support modern programming paradigms while maintaining efficiency and flexibility?"

Proposed Solutions

- **Solution 1:** Language-agnostic macro processor.
- **Solution 2:** Optimized performance with caching and parallel processing.

Solution 1: Language-Agnostic Macro Processor

What is it?

- A macro processor that can work across multiple programming languages.
- Uses Abstract Syntax Trees (ASTs) instead of simple text substitution.

Advantages

- Improves flexibility and portability.
- Reduces syntax errors compared to basic text-based macro expansion.

Solution 2: Optimized Performance with Caching

What is it?

- Uses caching and parallel processing to improve macro expansion speed.
- Reduces redundant computations during preprocessing.

Advantages

- Decreases compilation time for large codebases.
- Optimizes CPU and memory usage.

Implementation

- Use memoization to store frequently expanded macros.
- Implement multi-threaded macro expansion.
- Leverage GPUs for large-scale macro processing.

Core Idea:

- Memoization saves results of function executions to avoid redundant computations.
- This paper proposes applying memoization at compile-time instead of runtime.

What's New?

- Identifies memoizable functions during compilation.
- Generates optimized memoization wrappers integrated into the compiled code.

Key Features of Compile-Time Function Memoization

- **Broader Applicability:** Works for all types of functions, including user-defined ones.
- **Inlining for Efficiency:** Memoization wrappers are designed to be inlined, reducing overhead.
- **Handles Complex Scenarios:** Works with global variables, pointers, and constants.
- **Automatic Identification:** Suitable functions are detected automatically during compilation.

How is it implemented?

- Memoization is added as an optimization pass in the LLVM framework.
- **Step 1:** Analyze functions to identify memoizable candidates.
- **Step 2:** Generate memoization wrappers for each identified function.
- **Step 3:** Replace function calls with calls to memoization wrappers.

Memoization Wrapper:

- Checks a table for previously computed results.
- If not found, computes the result and stores it.

Benefits of Compile-Time Function Memoization

- **Performance Gains:** Avoids redundant computations and speeds up execution.
- **Reduced Overhead:** Inlining minimizes the cost of memoization.
- **Hardware Extension Proposal:** Suggests optional hardware support for further performance improvements.