Performance and Benchmark Analysis of a Bytecode Virtual Machine

January 7, 2026

1 Introduction

This report presents a detailed performance and benchmark analysis of a custom stack-based bytecode virtual machine (VM) and its associated assembler. The primary objective of this evaluation is to measure the efficiency of bytecode generation and execution across a diverse set of programs involving arithmetic computation, control flow, memory access, loops, and function calls.

The benchmarks aim to validate the correctness, scalability, and runtime efficiency of the virtual machine design under varying computational workloads.

2 Benchmark Methodology

Performance measurements were collected by embedding instrumentation directly within the assembler and the virtual machine runtime. All benchmarks were executed on a standard Linux environment.

The following metrics were recorded for each test program:

- **Instruction Count**: The number of static assembly instructions present in the source file.
- **Bytecode Size**: The size of the generated bytecode (`.byc`) file in bytes.
- **Assembler Time**: Time taken by the assembler to parse and translate the assembly source into bytecode.
- **Execution Time**: Time taken by the virtual machine to execute the bytecode program.

Assembler time was measured around the code generation phase, while execution time was measured exclusively around the VM execution loop, excluding bytecode loading and input/output operations. All timing values were recorded in milliseconds (ms).

3 Benchmark Test Programs

The benchmark suite consists of programs designed to exercise different components of the VM:

- **arithmetic.asm**: Evaluates arithmetic expression handling.
- **loop.asm**: Tests basic loop and branching behavior.
- **factorial.asm**: Implements an iterative factorial computation.
- **function_call.asm**: Verifies CALL and RET instruction correctness.
- **nested_loop.asm**: Exercises nested loops and control flow.
- **deep_nested_loop.asm**: Stress-tests repeated nested execution.
- **power.asm**: Computes exponentiation iteratively using loops and memory.

These programs collectively validate arithmetic correctness, memory consistency, control flow integrity, and call stack correctness.

4 Benchmark Results

Table 1: Performance Metrics of Benchmark Programs

Program Name	Instruction Count	Bytecode Size (Bytes)	Assembler Time (ms)	Execution Time (ms)
arithmetic.asm	16	48	0.167	0.006
deep_nested_loop.asm	16	48	0.095	0.006
factorial.asm	63	73	0.106	0.008
loop.asm	21	33	0.093	0.006
function_call.asm	6	18	0.194	0.004
nested_loop.asm	107	114	0.099	0.009
power.asm	120	83	0.203	0.007

5 Analysis

5.1 Assembler Performance

The assembler exhibits consistently low generation times across all benchmark programs. Even programs with a large number of instructions, such as `nested_loop.asm` and `power.asm`, require less than 0.21 ms for bytecode generation.

The relatively stable assembly times indicate near-linear scaling with respect to instruction count. This behavior confirms the efficiency of the assembler design and demonstrates minimal overhead per instruction.

5.2 Virtual Machine Execution Performance

All benchmark programs execute in under 0.01 ms, highlighting the efficiency of the virtual machine runtime. Programs involving complex control flow, such as nested loops and iterative computations, exhibit slightly higher execution times due to repeated dynamic instruction execution.

It is observed that execution time does not correlate strictly with static instruction count. For example, `power.asm` and `nested_loop.asm` incur higher execution costs due to loop iterations, despite comparable bytecode sizes. This demonstrates that runtime performance is primarily influenced by dynamic execution behavior rather than static program size.

5.3 Bytecode Density

The ratio of bytecode size to instruction count varies across benchmark programs, reflecting the variable-length instruction encoding used by the VM. Programs containing a higher proportion of operand-bearing instructions (e.g., PUSH, LOAD, STORE) exhibit larger bytecode sizes per instruction.

This confirms the correctness and efficiency of the chosen instruction encoding scheme, which combines compact single-byte opcodes with fixed-width operand instructions.