Report

1. Introduction

In this project, I have implemented different threading(direct, indirect, subroutine) models into virtual machines, and have a performance benchmark with different conditions(loop, size of code, the actual number of instructions executed).

2. Implement

2.1 Instruction Set

The virtual machines with three different threading techniques shared the same instruction set. From a functionality aspect, instructions are divided into four parts including arithmetic, memory controlling, flow controlling instructions, and debugging tools. All instructions are verified by the Gtest framework.

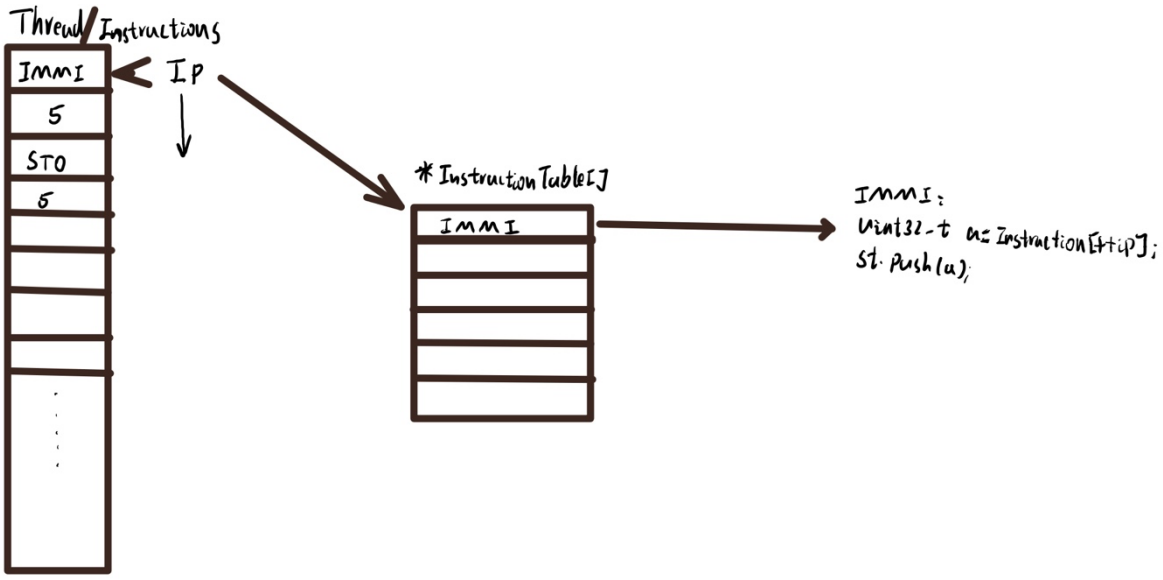
Arithmetic Instructions	
DT_ADD	Adds the top two unsigned integers on the stack.
DT_SUB	Subtracts the top unsigned integer from the second top integer.
DT_DIV	Divides the second top unsigned integer by the top integer.
DT_INC	Increments the top unsigned integer on the stack.
DT_DEC	Decrements the top unsigned integer on the stack.
DT_SHL	Performs a left shift on the second top unsigned integer by the number of bits specified by the top integer.
DT_SHR	Performs a right shift on the second top unsigned integer by the number of bits specified by the top integer.
DT_FP_ADD	Adds the top two floating-point numbers.
DT_FP_SUB	Subtracts the top floating-point number from the second top floating-point number.
DT_FP_MUL	Multiplies the top two floating-point numbers.
DT_FP_DIV	Divides the second top floating-point number by the top floating-point number.
Memory Control	
DT_LOD	Loads an unsigned integer from the specified memory offset into the stack.
DT_STO	Stores the top unsigned integer into the memory at the specified offset.
DT_IMMI	Pushes an immediate unsigned integer value onto the stack.
DT_STO_IMMI	Stores an immediate unsigned integer into the memory at the specified offset.
DT_MEMCPY	Copies a block of memory from a source to a destination address.
DT_MEMSET	Sets a block of memory to a specified value.
Flow Control	
DT_END	Clears the stack and resets the instruction pointer.
DT_JMP	Unconditionally jumps to a specified instruction.
DT_JZ	Jumps to a specified instruction if the top unsigned integer is zero.
DT_JUMP_IF	Jumps to a specified instruction if the condition is true.
DT_GT, DT_LT, DT_EQ, DT_GT_EQ, DT_LT_EQ	Comparison instructions that push 1 (true) or 0 (false) based on the comparison result.
DT_CALL	Calls a function at a specified address.
DT_RET	Returns from a function call.
Debugging Tools	
DT_SEEK	Assign the top of the stack to “debug_num”.
DT_PRINT	Prints the top unsigned integer on the stack.
DT_FP_PRINT	Prints the top floating-point number on the stack.
DT_READ_INT	Reads an integer from standard input and stores it at a specified memory offset.
DT_READ_FP	Reads a floating-point number from standard input and pushes it onto the stack.

2.2 Thread Model

In this project, I mainly implemented three techniques into virtual machines which are the direct threading model, the indirect threading model, and the subroutine model.

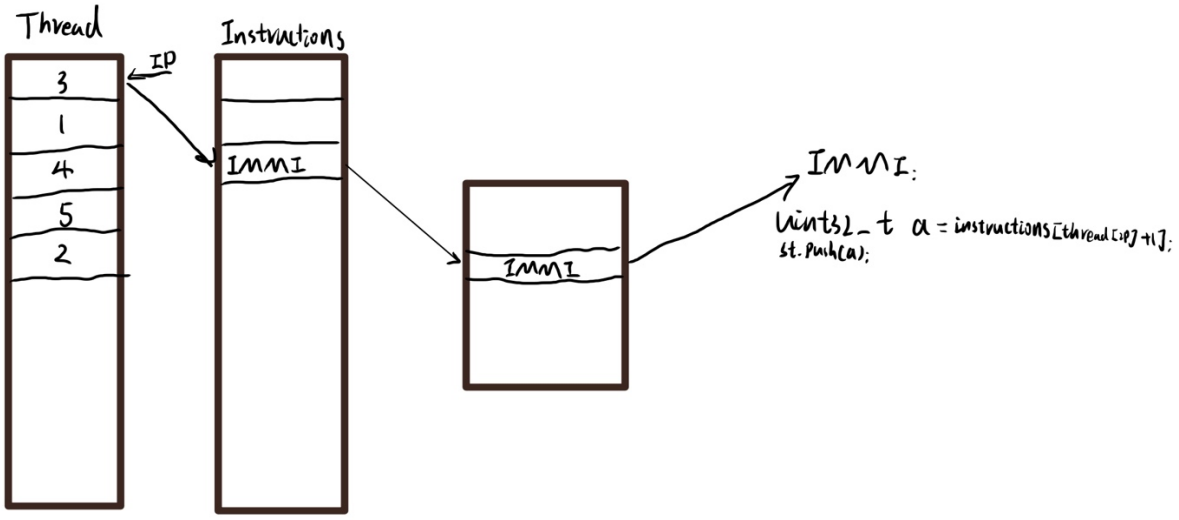
2.2.1 Direct threading model

In the Direct threading model, the instructions that need to be executed are stored in the thread. I take user input as the thread when parsing the instruction code and let the virtual machine sequentially execute the code inside.



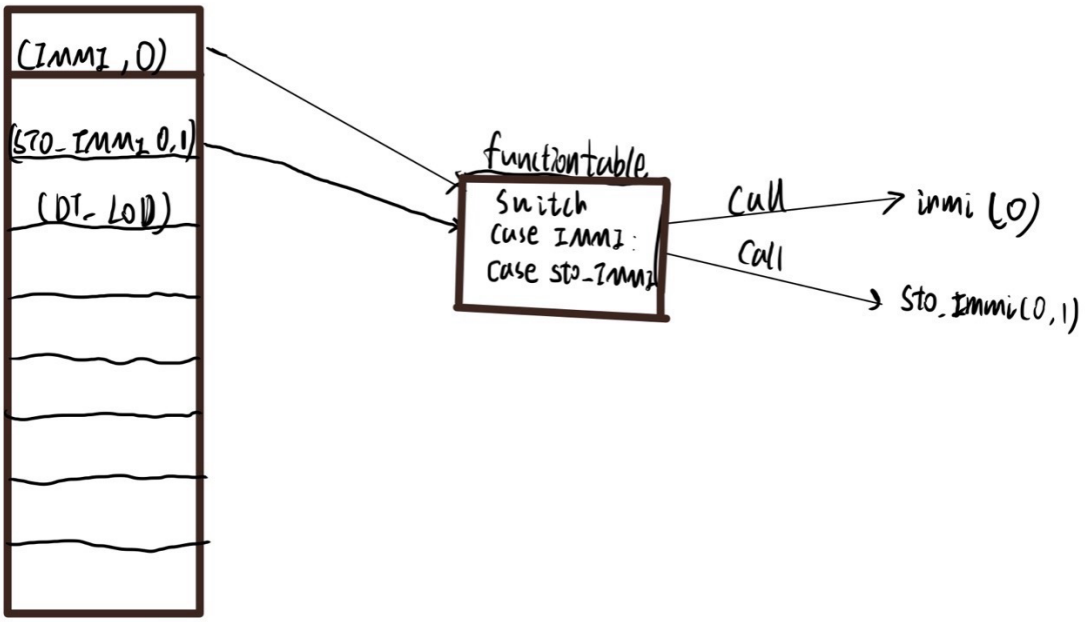
2.2.2 Indirect threading model

In the Direct threading model, the instructions are not recorded in the thread, but the address of memory instead. Technically, the Indirect threading model supports the disorder of instruction sequence, however, to make the compiler simple and make sure that the same program can be run in different threading models, all the instructions are loaded in order, and I added a built-in module for converting the layout of code into a format that can be executed by the indirect threading model. This may introduce additional performance overhead



2.2.3 Subroutine threading model

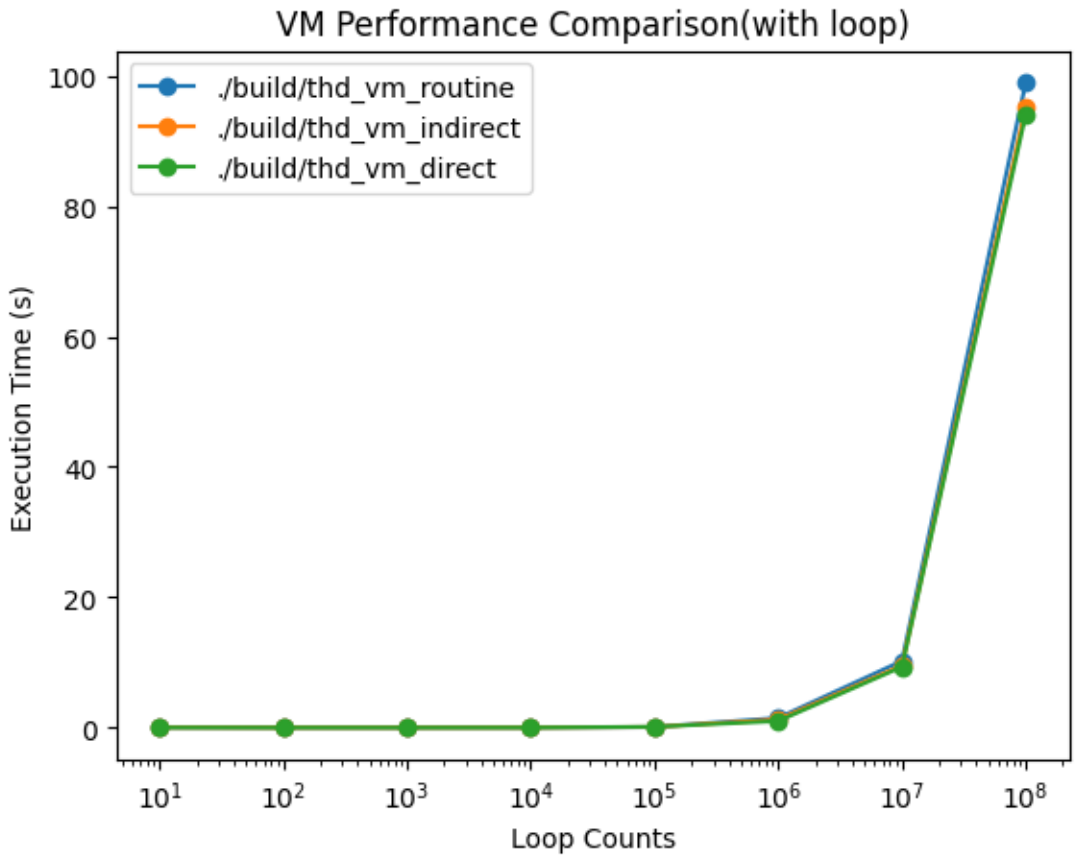
In the subroutine threading model, a set of operands and operators are considered as a function. Similar to the Indirect threading model, I also added a converting model to transfer the layout of the code. This may also introduce additional performance overhead



3. Result:

3.1 Test code:
"DT_IMMI,0,DT_STO_IMMI,0,1,DT_LOD,0,DT_ADD,DT_LOD,0,DT_INC,DT_STO,0,DT_LOD,0,DT_IMMI,{loop_count},DT_GT,DT_JZ,5,DT_PRINT,DT_END"
Task: To measure the performance when the size of the code is small(To reduce the impact of converting code layout for indirect and subroutine models)
Functionality: sum up from 1 to 10, 100, 1000, 10000, 100000, 1000000, 10000000, 100000000
Equivalent C code:

```c
int i=0;
int sum=0;
for(int i=0;i<N;i++){
    sum+=i;
}
```

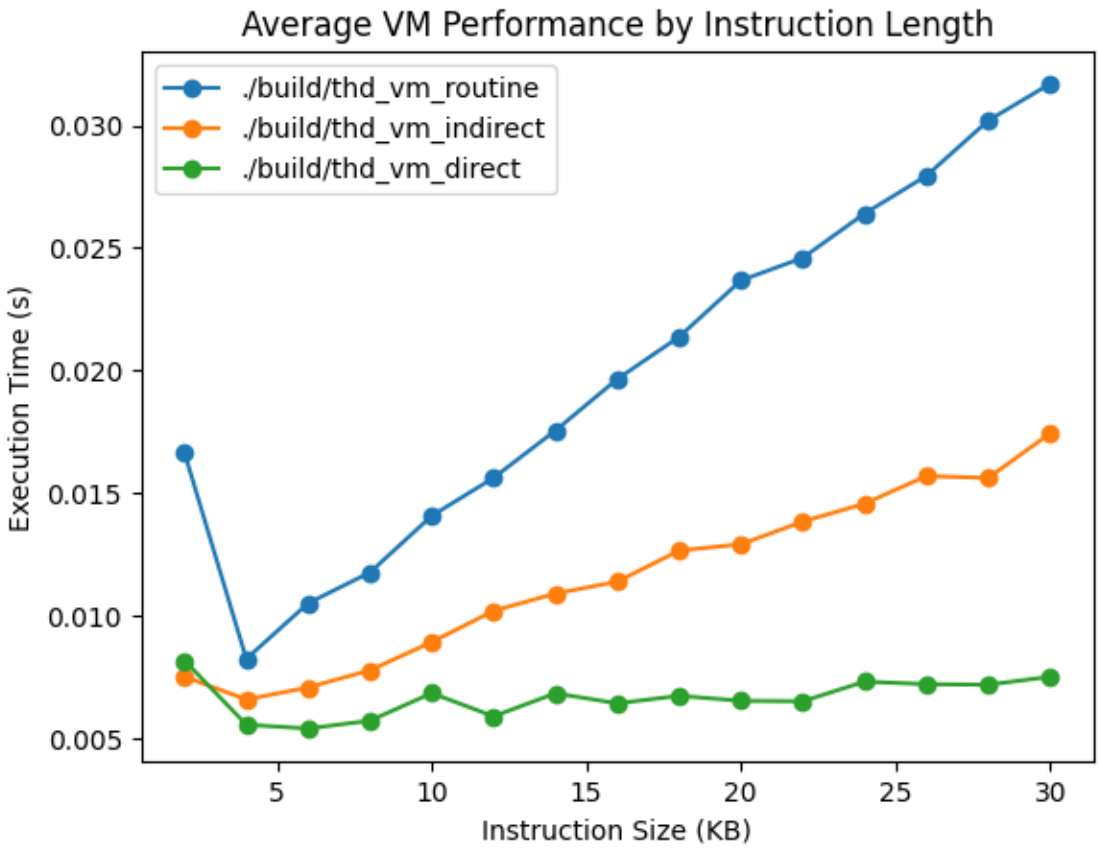


This figure indicates that, when the sequence of code is short, the performance of different threading models is similar.

3.2 Test Code:

"DT_IMMI,1," * num_instructions(make size up to 2kb,4kb,6kb,8kb,10kb)

Task: To measure the impact of performance loss due to code converting.



The chart clearly demonstrates a linear increase in virtual machine (VM) runtime in relation to the lengthening of code. Among the threading models evaluated, the subroutine threading model exhibits the most rapid increase in runtime, suggesting a heightened sensitivity to code length. This is followed by the indirect threading model, which shows a notable but less steep increase. In contrast, the direct threading model displays a much more gradual change, indicating a relatively lower impact from code size variations. This trend underscores the direct threading model's resilience to increases in code length, in comparison to the more code length-sensitive subroutine and indirect models.

Test Code:

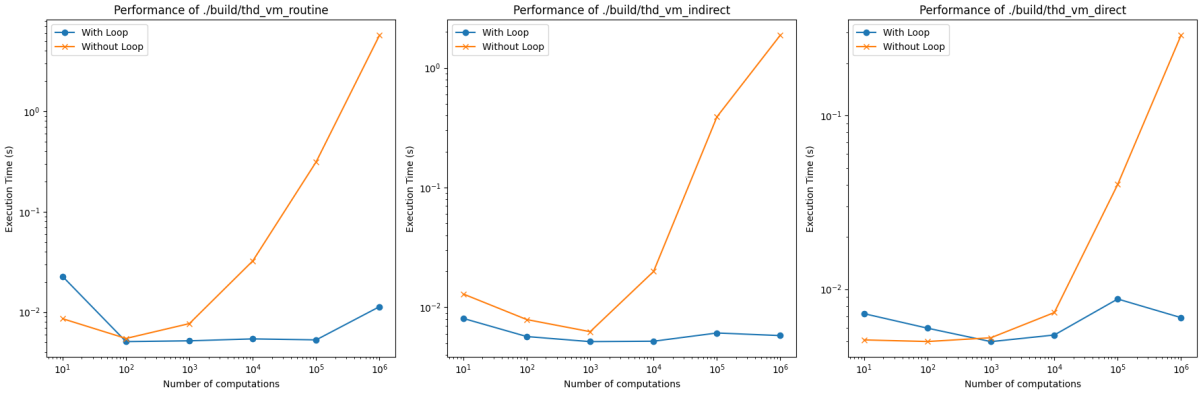
"DT_IMMI,0,DT_STO_IMMI,0,1,DT_LOD,0,DT_INC,DT_STO,0,DT_LOD,0,DT_IMMI,{count},DT_LT,DT_JZ,5,DT_END"

"DT_IMMI,1,"+"DT_INC," * count + "DT_END"

Task:

The experiments conducted provide insights into how code length variably influences different threading models. To further quantify this impact, an additional test project is introduced. However, it's crucial to understand that when dealing with extensive code, the execution efficiency of a loop can be comparable to, or at least in the same order of magnitude as, that of a non-loop structure. For instance, the task of summing numbers from 1 to 100 can be executed either through a loop or by generating a lengthy code sequence like 1+1+2+...+100. This approach allows for a rough comparison of how code length affects execution speed. The significance of this comparison lies in its ability to elucidate the performance dynamics of different threading models under varying code lengths, thereby

providing a more nuanced understanding of their operational efficacies.



The charts illustrate the varying performances of three distinct threading models, both with and without a loop, when executing the same function. It's observed that the performance without a loop tends to be superior with a lower count of operations, a result attributed to the jump statements' significant overhead and more rapid sequential code execution. However, as code length expands, the non-loop variant necessitates more frequent memory data retrieval, leading to a reduced cache hit rate and ultimately being outperformed by the loop-inclusive version. Notably, the juncture at which the loop version surpasses varies across the three VMs. Both the subroutine and indirect threading models exhibit comparable performances at 100 computations, regardless of loop presence, while the direct threading model shows a similar trend at 1000 loops. This aligns closely with previous analyses comparing code length's impact on various VMs' performance.

4. Conclusion

In scenarios where the code length is minimal, the performance discrepancies among the three threading models are negligible, with the subroutine threading model exhibiting superior efficiency, particularly at higher loop counts (refer to test one). Conversely, in cases of extensive code, the direct threading model emerges as markedly more beneficial (refer to test two). It is advisable to implement loops in situations where an instruction is executed in excess of 1000 times (refer to test three).