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.

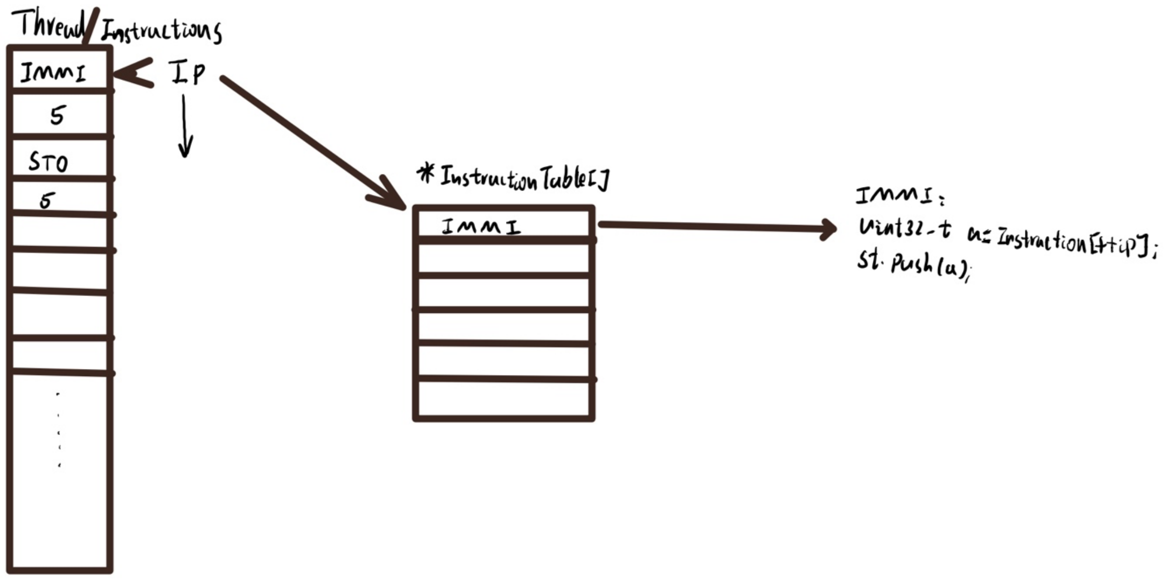
Arithmetic Instructions	
DT_ADD	Adds the top two unsigned integers on the stack.
DT_SUB	Subtracts the top unsigned integer from the second top integer.
DT_DIV	Divides the second top unsigned integer by the top integer.
DT_INC	Increments the top unsigned integer on the stack.
DT_DEC	Decrements the top 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.

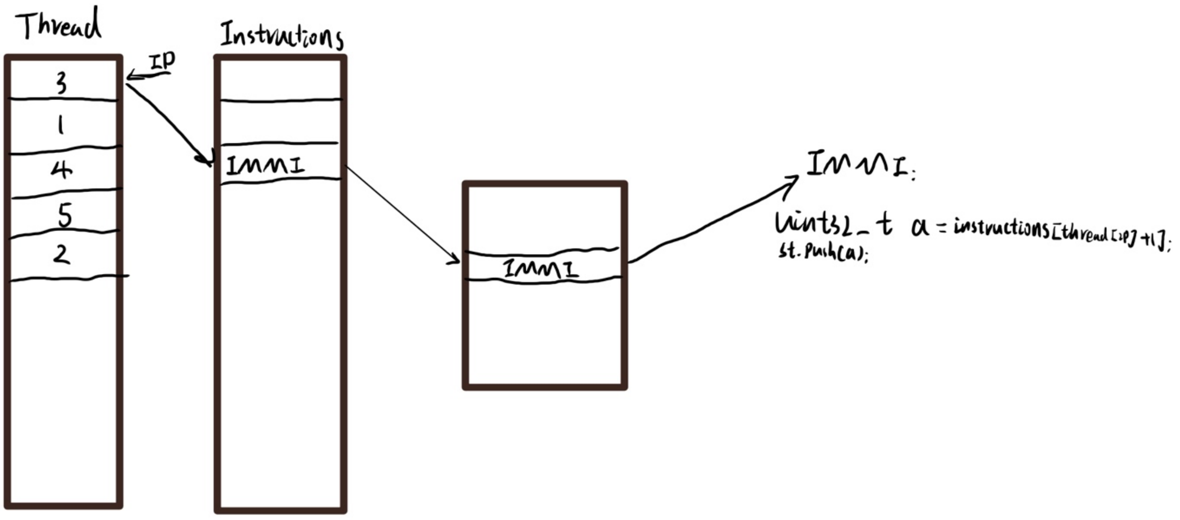
2.2.1 Direct threading model

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.



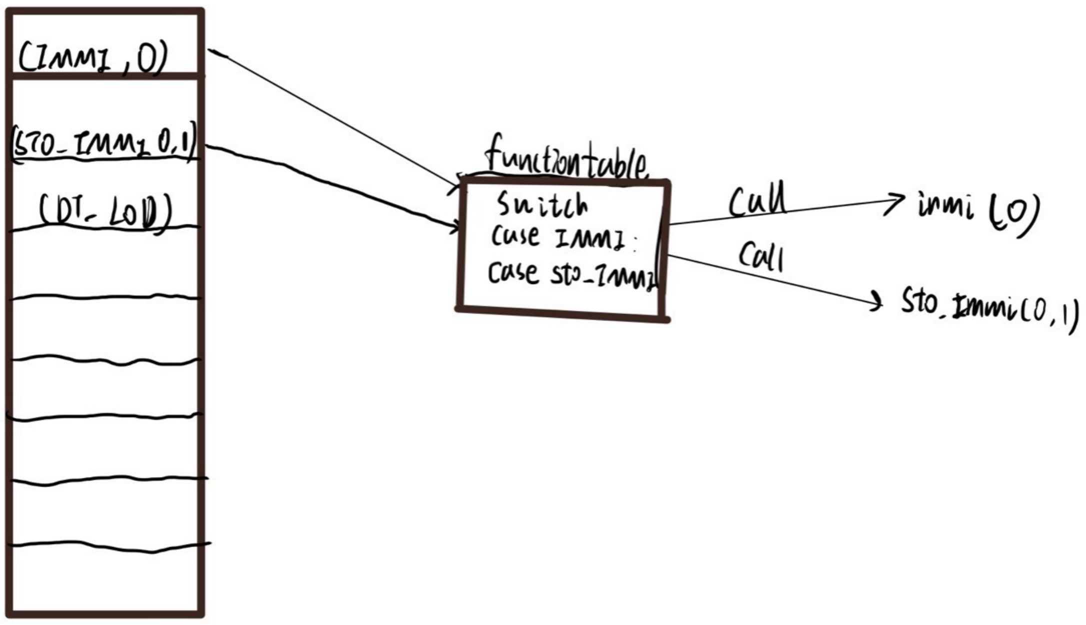
2.2.2 Indirect threading model

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



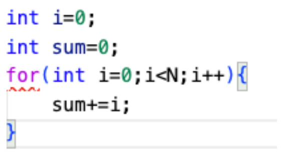
2.2.3 Subroutine threading model

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

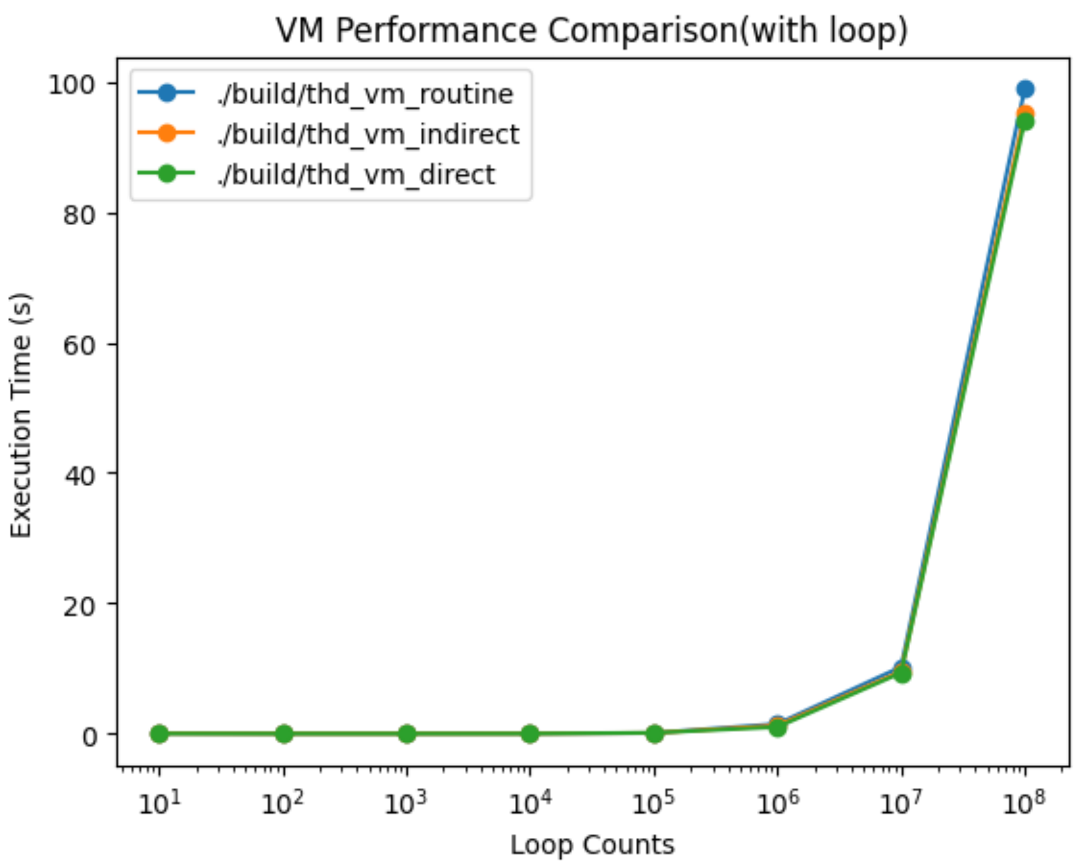


3. Result:

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:



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

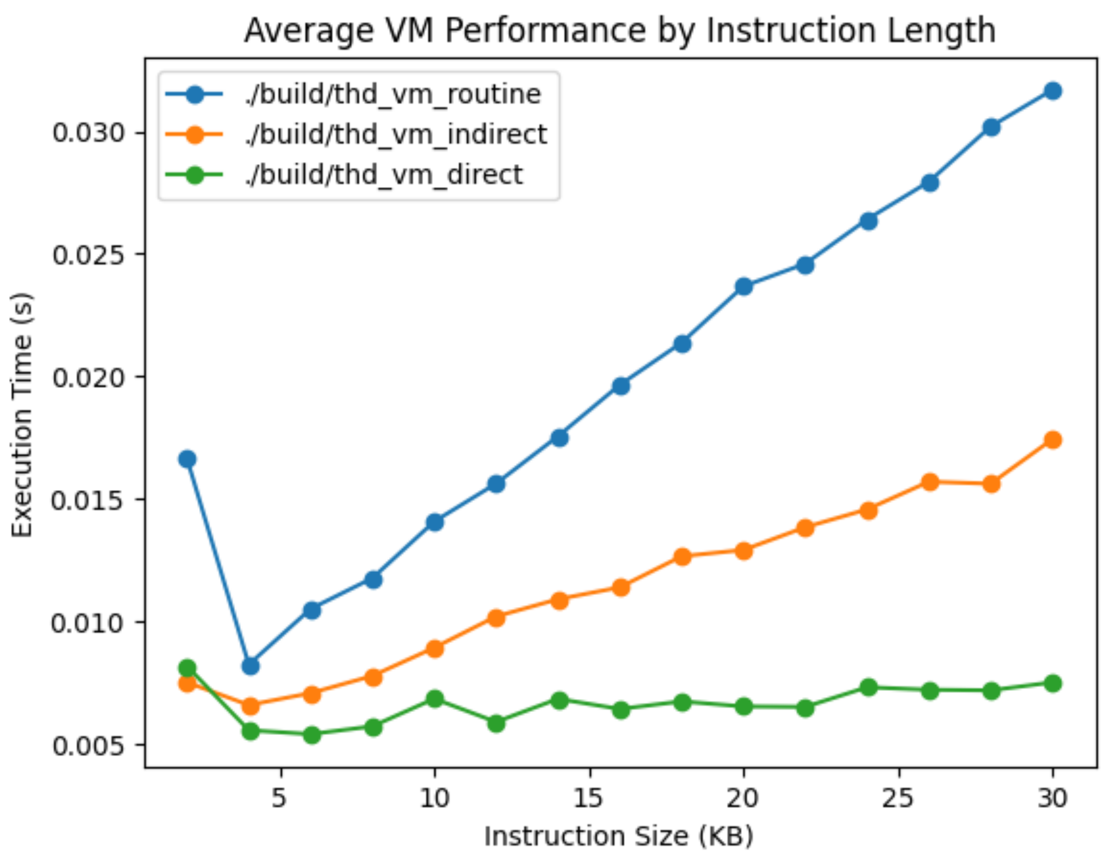


This figure indicates that, when the sequence of code is short, the performance of different threading models is similar.

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.



This chart shows that as the code length increases, the VM runtime increases linearly. The subroutine threading model has the fastest increasing trend, followed by the indirect threading model, and the change of the direct threading model is much slighter than others, which indicates that it is less impacted by the size of the code.

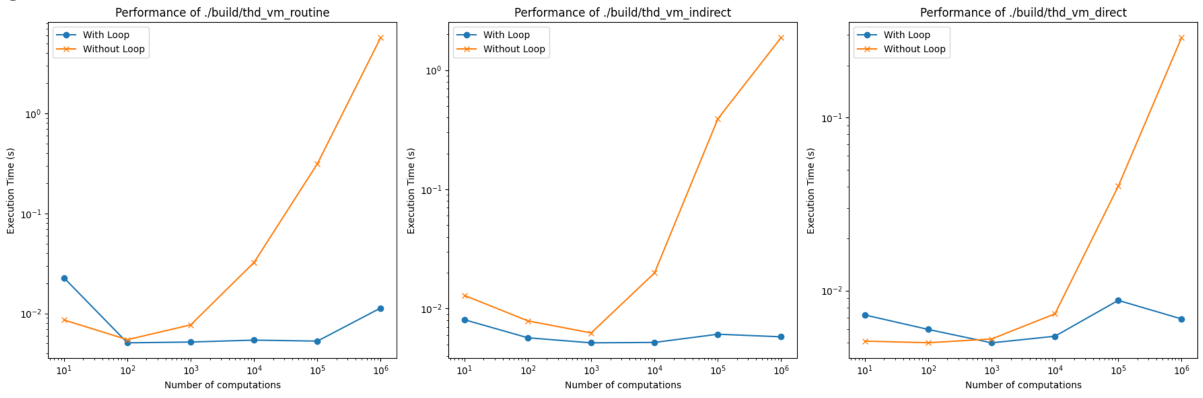
Test Code：

"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:

Based on the above experiments, we can already get a feel for how code length affects different Threading models differently, and below I've added a test project to quantify such an impact. But first we need to clarify the concept that when the code is long enough, very often the code execution of a loop can be myopically equal to that of a non-loop, or at least they are of similar order of magnitude. For example, adding from 1 to 100 can be accomplished with a loop, or it can generate a code of 1+1+2...+100. By doing this, we can roughly compare the effect of code length on execution speed.



These three charts show how the performance of three different Threading models with and without loop varies when accomplishing the same function. We can see that the average performance without a loop is higher when the number of accumulations is low, which can be explained by the high overhead of jump statements and the faster sequential execution of the code, however, as the length of the code increases, the one without loop needs to read data from memory more frequently, which leads to a lower cache hit rate, and the performance is rather exceeded by the one with loop. However, the time to exceed is not the same on the three VMs, as can be seen, subroutine and indirect threading models are quite close with and without loop at 100 computations, while direct threading is close at 1000 loops. This is close to the length when we compared the effect of code length on the performance of different VMs earlier.

4. Conclusion

When the code size is small, there is little difference between the three schemes, and the subroutine threading model even has higher performance when the number of loops is high (see test one), however, when the code size is large, the direct threading model is more advantageous (see test two). Try to use loops when the instruction is executed more than 1000 times (see test three).