Virtualization Techniques WS 22/23 (Weidendorfer)
Homework 2: Efficient Execution of Bytecode

In this task, you will develop efficient ways to execute an instruction stream from a very simple virtual ISA. The plan is to have weekly short discussions of each of task 1 - 6 in the lectures starting at Nov 9, 2022. You may use any programming language you want, but the input data for comparable results is given in C – you will need to either port to your language of choice, or capture the input generated by the C code as input.

For one task below, we want to have access to Hardware Performance Counters for more detailed measurement. A nice simple tool for that is "perf" on a recent Linux distribution. This does **not** work in a VM like VirtualBox, so you need a native Linux installation. E.g. on Ubuntu, the tool "perf" can be found in package "linux-tools-common", binary is "/usr/bin/perf".

Our virtual machine has 3 registers: IP (instruction pointer), A (accumulator), and L (loop counter). IP can hold 32-bit positive addresses, A and L both have type 32-bit signed integer. Register IP is initialized to 0 at start, but both A and L can be set (e.g. by "boot" parameters) to arbitrary values by the user at start. Furthermore, the machine has the following 6 instructions, encoded as 1 byte per instruction (= the opcode):

- HALT (opcode 0): stop execution
- CLRA (opcode 1): set content of register A to 0
- INC3A (opcode 2): increment register A by 3
- DECA (opcode 3): decrement register A by 1
- SETL (opcode 4): copy value of register A to L
- BACK7 (opcode 5): decrement L; if value of L is positive, jump back by 7 instructions (i.e. loop body is 6 one-byte instructions and the BACK7 itself). Otherwise fall through to next instruction

On the VT website there is a C code provided ("gen.c") to (pseudo-)randomly generate instruction sequences for this "virtual machine" into a C char array (as well as start values for register A/L). It randomly mixes opcodes 1 - 5, and it finishes the instruction stream with HALT. For the following tasks, we assume that the value of PC is an index into this C array, and it specifies the next instruction to execute. The generator function "init()" has 3 inputs: (1) code size in bytes = instructions, (2) seed of pseudo random number generator, (3) relative probabilities of the instructions with opcodes 1 – 5 in the generated bytecode stream. Parameters 4 and 5 are pointers to values to be initialized by "init()" for start values of register A and L.
For comparison of performance between solutions, it is important to compare the same instruction sequence with same start values of registers. For this, we will use the result of calling init() with the following inputs:

- scenario 1: size 10000, seed 1, probabilities 0/1/0/0/0 (for opcodes 1/2/3/4/5)
- scenario 2: same as 1, but with probabilities 1/1/1/0/0
- scenario 3: size 10000, seed 1, probabilities 1/9/1/5/5
- scenario 4: same as 3, but size 50000

Execution time of a scenario is the wall clock time required to run the instruction sequence once. For fairness with simple interpretation (task 1), any time spent for dynamic optimization in the following tasks (such as pre-decoding or code generation) has to be added to the execution time: spend optimization effort wisely!

Task 1: Basic Interpretation

- define a data struct able to hold the machine state
- write an interpreter for this ISA which is able to execute a program given as a byte array and a given machine state (as parameters to the interpreter function), returning at HALT
- measure the average time required by your interpreter for executing one instruction, averaged across the input scenarios given above, both as absolute time and number of clock cycles.
- what is the best timer source to use for measurement?
- how to get clock cycles from time? (Hint: what is "turbo boost"?)
- how to get stable measurement results? Hint: measure the elapsed wall-clock time over so many iterations of a full interpretation run such that the time span becomes at least a few seconds
- the difference in execution time between input scenarios 1 and 2 partly relates to branch prediction behavior. Using "perf stat <interpreter>", you can find out the number of branch mispredictions. How large is the cycle penalty of one misprediction for the CPU in your laptop on average?

Task 2: Interpreter Optimization

Use different strategies for improving the performance of your interpreter (from chapter 2 of the lecture):

- indirect threaded interpretation
- predecoding with direct threaded interpretation
- reducing the number of jumps by using "superevents" (combine 2 original instructions in a row).

How much better do you get?

Task 3: Code Generation for opcodes 0-4 (without control flow changes)

In this task, we ignore BACK7 instructions in programs. Thus, the “guest program” is a single basic block.

- what is a good mapping of instructions to x86 instructions?
- what is a good mapping of state of our virtual machine to x86 state?
- directly generate an x86 instruction stream in memory from the “guest programs” in the given input scenarios, ignoring instruction BACK7. Measure the execution time. How fast can you get?
  Hints: for this, you need to make allocated memory executable via "mprotect" (Linux). Find the sequence of machine codes for the function prolog/epilog on your architecture (e.g. using gdb's "disass /r" command on compiler generated code), and generate this before/after the generated x86 code, to be able to call the code via a function pointer from C. To check your generated code, in the debugger gdb, after code generation, use e.g. "x/5i address" to show 5 instructions at address
- which optimizations are useful to do within a BB, taking the specific instructions into account? Use these to improve your generator. Pre-calculation of final result for initial reg settings not allowed!

Task 4: Code Generation for Loops (including BACK7)

Programs actually can contain multiple (dynamic) basic blocks (BB), and you need to be able to generate code for multiple BBs, which are to be called from an emulator loop.

- what is a good function signature to call into generated code of a BB from an emulation loop? What has to be done in the prolog and epilog of code generated for a BB?
- pre-generate translations of all BBs existing in a program using your code generator from Task 3. Write an emulator loop able to call these generated translations for execution of a given program. How much faster do you get than your interpreter variants?
- instead of pre-generation of translations for all BBs, modify your emulator loop to trigger translation on demand. What is the overhead?
- now write an interpreter able to execute one BB. Modify your emulator loop to trigger code generation **after one interpretation** of an BB. How much less code does this generate? Does this help performance?
- from Task 3, you should have a simple generator and one doing optimizations. Modify your emulator loop to enable **optimization stages**. Does this help performance?

Task 5: Optimized Code Generation (1): Chaining

- For chaining, we want to be able to patch the end of generated code of BBs to directly jump to following BBs. What needs to be modified in your code generator?
- Write a function able to chain BBs, and modify your emulator loop to do chaining if possible. How many chainings are done in the given scenarios? How much faster do you get?

Task 6: Optimized Code Generation (2): Superblocks

- Now we want to build superblocks by appending the loop body after a BACK7 instruction into one generated piece of code. What needs to be changed in the code generator? Do you get faster?
- A better strategy for superblocks here is e.g. 2x unrolling of loop bodies before BACK7 instructions. When does it make sense to trigger generation of such superblocks? Implement that strategy.