Rum Profiling Assignment

Ben Keeler and Ryan Brooks

Help Received:
We utilized the slow Rum provided by professor Daniels and we asked one or two questions on Edstem.

Hours Spent:
We spent approximately 5 hours analyzing the problems posed and 5 hours solving them.

Lab Notes Overview:
The initial rum was clocked in at 33 seconds for midmark and 804 seconds for sandmark. The first initial gain was from compiling in release mode. Without the additional runtime checks and assertions from debug mode, it improves the speed greatly. It got down to 8 seconds for midmark after this adjustment. Next, an extra conditional in the load program instruction helped avoid unnecessary computations which brought the midmark time all the way down to 2.5 seconds. Our next monumental gain was made from changing the underlying data type of the memory representation. It was originally a hashmap of vectors but we made the adjustment to a vector of vectors. This was due to the fact of wanting to access the vectors by index compared to a key which makes for faster lookup times. It brough midmark down to 0.33 and sandmark to 6.91. Lastly, we made another gain due to the way we were accessing the memory elements in the load and store functions. We initially were using the .get function to access the values in the vector of vectors but we made the change to directly index into it. With this implementation we are assuming the indices are always valid. We also got rid of a call to unwrap because there was no more option which is returned from the get function. This brought down midmark to 0.285 and sandmark to 6.104. Although this improves the speed, it gives the program one more way to potentially fail because it does not have the option to handle a none case compared to the alternative. We also added a benchmark which was in the middle of sandmark and midmark in terms of overall instructions. It was a partial solution to advent.umz.
To put it in context:
Midmark: 85,070,522 instructions
Sandmark: 2,113,497,561 instructions
Adventure solution: 777,658,728 instructions

With the advent.umz solution we clocked in at around 2.10 seconds. Our midmark time ended up at around 0.28 seconds. Sandmark with roughly 6.35 seconds as all were timed with hyperfine.

Assembly Analysis

Our most expensive function was the decode function in machine.rs. This function is responsible for taking a u32 instruction and returning an instruction object. It first matches the opcode that has been right-shifted by 28 and bitwise anded with 0b1111. It then matches the value for each instruction form 0-13. Then an instruction object is created. It then matches if the opcode is a load value or not. It updates the instruction fields for ra, rb, rc, or ra and value depending on the case.

The assembly produced by this function is as follows: there are some initialization steps such as saving registers eax, rdi, and rcx on the stack, then the opcode decoding starts. We can see bitwise operations including shr, and, and shl in the assembly. There are also conditional jumps based on the opcode value. We can see a jump table implemented with a series of mov dword and jmps that represents the pattern matching for the opcode. After that there's additional shifts and bitwise and operations for the load value match case.

In our analysis of the assembly code we found that while generally optimized, there were a large number of jump calls as well as memory accesses and stores. While it makes sense that there would be additional stores and loads, the number of jumps might be able to be lessened by improving the way we handle matching and reusing opcodes throughout the program. Another improvement could be creating constants for the mask and shift amounts or replacing large shifts such as inst.value = (instruction << 7) >> 7 with instruction & 0x1FF. In summary, this function did not rely on any other functions and mostly consisted of bit shifts and other bitwise operations. There was decent assembly produced; therefore, we think that general handling of opcodes and matches might lead to fewer jumps in the assembly which might be more efficient.

Final Performance

Compared to where we started the final version of the program has been optimized in many big ways. We could tell we were getting miniscule improvements for a few of our last changes (except for the last change we documented). Moreover when analyzing the code in Samply, we found that the most expensive functions – that being decode and get_instruction were – were used in only one place in the program and were required for every instruction. The functions themselves, and moreso for get_instruction were very optimized. We felt the same for the other functions as well. That led us to feel that we were confident and in how much our program had been optimized compared to where it started.