Data Table:

Benchmark	*Time (ms)*	*Instructions*	*Rel to start*	*Rel to prev*
big	23129	N/A	1.000	1.000
small	1509	2.69×10^{10}	1.000	1.000
No improvement, starting point.				
big	21032	N/A	0.995	0.995
small	852	2.53×10^{10}	0.565	0.565
Compiled with optimization turned on and linked against -lcii-O1. Before, the program was run with no optimization flags, meaning the code was compiled using exact translation. The -O1 is the first tier flag of compilation optimization with dead code eliminated and simple instruction combining.				
big	20534	N/A	0.888	0.976
small	832	2.52×10^{10}	0.551	0.975
Compiled with optimization turned on and linked against -lcii-O2. Before, the program was run with the first tier -O1 optimization flag. The code is now run with the next tier compilation optimization flag, -O2, which is more aggressive.				
big	17350	N/A	0.750	0.845
small	672	1.59×10^{10}	0.445	0.808
Removed Bitpack_getu in unpack (for opcode/registers). Previously, we used the Bitpack_getu method from the provided Bitpack module, but we instead moved to bit manipulation to get the opcodes and registers.				
big	16579	N/A	0.717	0.956
small	669	1.59×10^{10}	0.443	0.996
Used stat to determine filesize to stop asserting each byte is EOF. Before, this assertion for every byte in the file was costly. Thus, we realized we could find the file size at the beginning, and used a single assert to make sure it is divisible by 4.				
big	17141	N/A	0.741	1.034
small	663	1.59×10^{10}	0.439	0.991

Removed Bitpack_newu when making the instructions. Instead, we relied on raw bit manipulation since repeatedly calling the Bitpack_newu function from the given Bitpack module was very costly.				
big	16003	N/A	0.692	0.934
small	633	1.34×10^{10}	0.419	0.955
Removed Bitpack_getu in the load value instruction helper function. Instead, we relied on raw bit manipulation since repeatedly calling the Bitpack_getu function from the given Bitpack module, especially on the common load value instruction, was very costly.				
big	15738	N/A	0.680	0.983
small	608	1.29×10^{10}	0.403	0.961
Removed Seq_length assertion in segmented store and segmented load instruction helper functions. Previously we were asserting for segmented loads/stores that were out of bounds done by calling Seq_length. However, these repeated Seq_length function calls were costly and since these errors were failure modes, we removed them.				
big	16025	N/A	0.693	1.018
small	621	1.29×10^{10}	0.412	1.021
We believed that the interactions from module to module could have hindered performance thus we moved everything into one single file. However, this seemed to not really have much improvement and even a negative effect.				
big	15994	N/A	0.692	0.998
small	627	1.29×10^{10}	0.416	1.010
We believed that making all of the functions in this new file static inline would have improved performance since there shouldn't have been function call overhead and the code would be placed directly where it's called. However, there seemed to really be no effect on performance.				
big	15418	N/A	0.667	0.964
small	593	1.28×10^{10}	0.393	0.946
We replaced all of the uses of uint64_t's into uint32_t's. This also meant that we were able to remove the wrapping logic that we explicitly had to do for the add and multiplication instruction logic. We previously used uint64_t's since we thought they had to be used for the Bitpack module. Since getting rid of the function calls from that module, turning them into uint32_t's resulted in less memory usage and better cache performance.				
big	14600	N/A	0.631	0.947

small	576	1.25×10^{10}	0.382	0.971
Changed freeID's stack to a C array. Since the function calls to the stack module were costly, we changed the data structure that represented the freeID's in our memory struct to a dynamic C Array.				
big	10199	N/A	0.441	0.699
small	399	9.37×10^{9}	0.264	0.693
Changed segments sequence to a C array. Since the function calls to the Hanson sequence module were costly, we changed the data structure that represented the segments in our memory struct to a dynamic C Array. Also made storeProgram function return the whole memory struct rather than just the individual segments data structure (that went from a Hanson sequence to a C array).				
big	10061	N/A	0.435	0.986
small	395	9.24×10^{9}	0.262	0.990
Optimized segmented store, load, and load value instruction code every so slightly. This essentially meant compacting the multi-line code for each function into one or two lines by consolidating bit manipulation or removing unnecessary assertions.				
big	7546	N/A	0.326	0.750
small	297	7.63×10^{9}	0.197	0.752
We removed the unpack function from our file which meant moving the unpacking logic into the cycle function. Because of the reduction in passing my reference and dereferencing for every single instruction, our performance improved.				
big	6370	N/A	0.275	0.844
small	253	5.75×10^{9}	0.168	0.791
We removed all the helper functions for the instructions and put them directly within each respective switch case. The reduction in passing by reference and dereferencing improved our performance.				
big	6057	N/A	0.262	0.951
small	239	5.41×10^{9}	0.158	0.945
Moved and stored the mem->segments deference outside of the while loop in the cycle function to drastically reduce the number of dereferences.				
big	5827	N/A	0.252	0.962

small	227	5.24×10^9	0.150	0.950
We store a local pointer to segment 0 outside of the while loop within the cycle function to avoid the repeated dereferencing.				
big	5480	N/A	0.237	0.940
small	214	5.02×10^9	0.142	0.943
Created C Array in memory struct to hold the freed segments rather than actually freeing them. We noticed the bottleneck of freeing seen in qcachegrind which was the result of the map and unmapping excessive memory allocation logic. Thus, instead of actually freeing segments, we stored them in this array and potentially reuse them.				