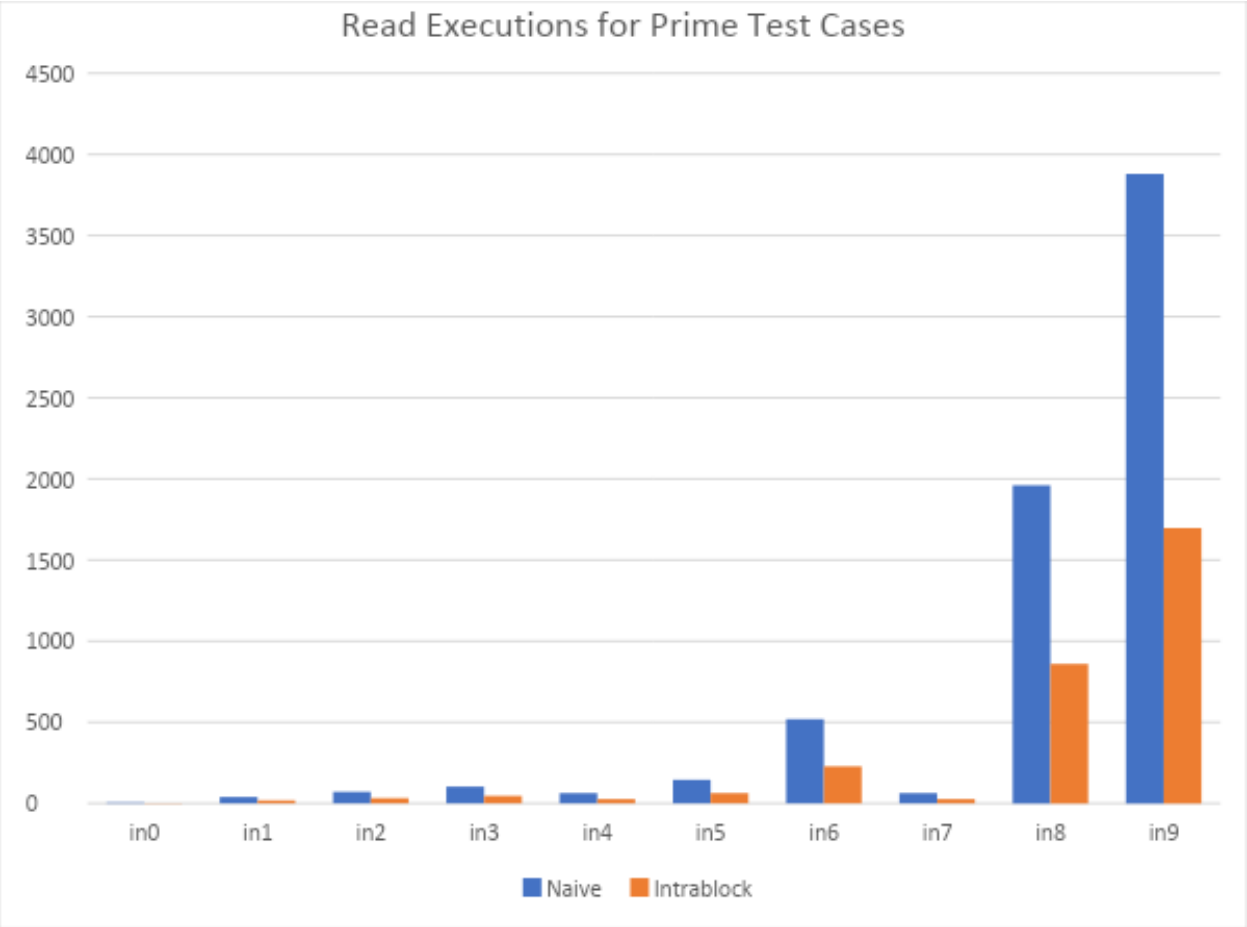
Performance Evaluation

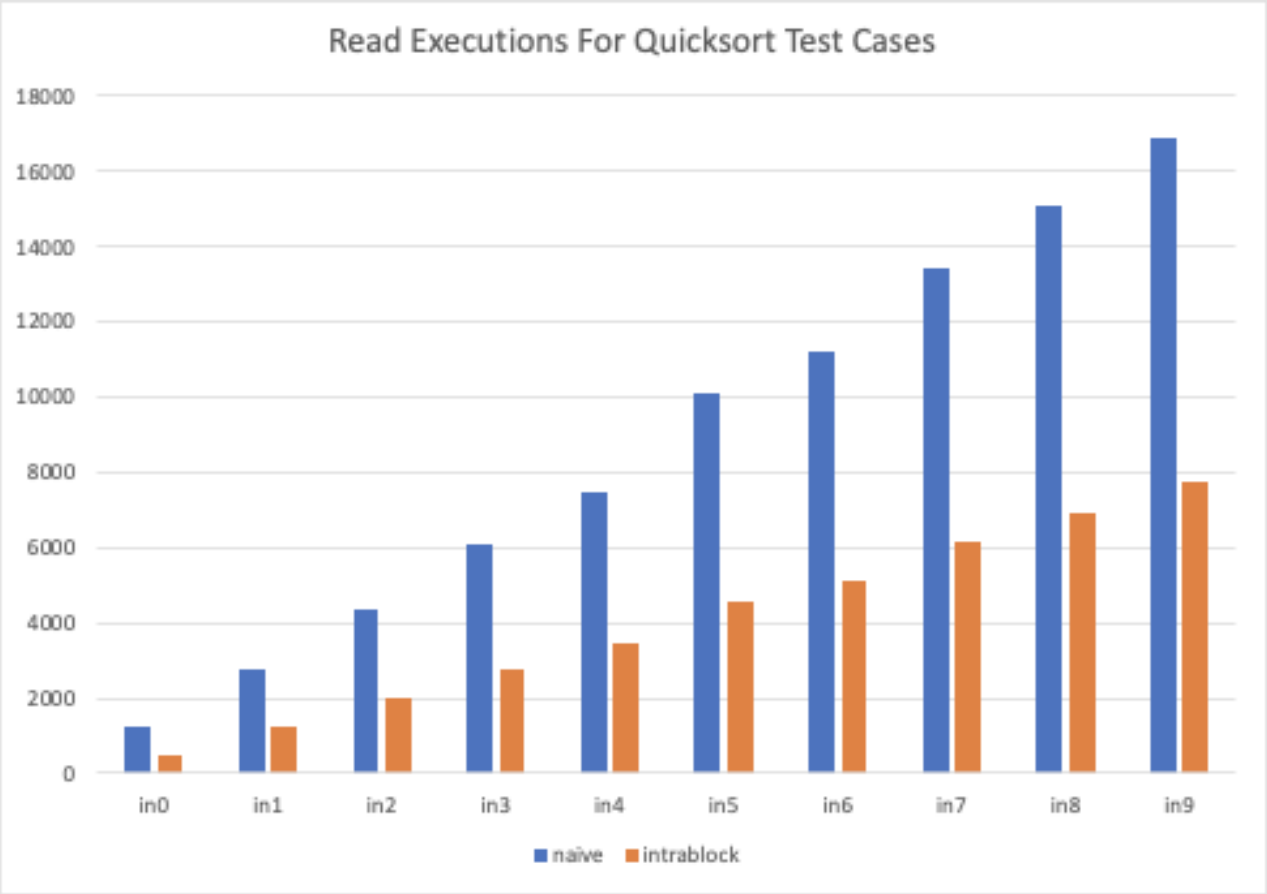
Naïve Prime	reads	writes	branches	other	Total:
0.in	5	25	3	55	88
1.in	39	54	15	108	216
2.in	71	83	27	160	341
3.in	103	112	39	212	466
4.in	63	77	26	153	319
5.in	145	151	57	286	639
6.in	519	489	195	888	2091
7.in	63	77	26	153	319
8.in	1959	1794	735	3220	7708
9.in	3879	3534	1455	6348	15216

Intra-block Prime	reads	writes	branches	other	Total
0.in	2	23	3	55	83
1.in	18	47	15	108	188
2.in	32	72	27	160	291
3.in	46	97	39	212	394
4.in	26	67	26	153	272
5.in	63	131	57	285	536
6.in	228	422	195	888	1733
7.in	26	67	26	153	272
8.in	858	1547	735	3220	6360
9.in	1698	3047	1455	6348	12548



Naïve Quicksort	reads	writes	branches	other	Total:
0.in	1256	999	179	1362	3796
1.in	2747	2162	405	2982	8296
2.in	4381	3416	647	4687	13131
3.in	6065	4702	890	6431	18088
4.in	7488	5833	1107	8019	22447
5.in	10075	7654	1449	10284	29462
6.in	11213	8636	1631	11725	33205
7.in	13415	10228	1936	13757	39336
8.in	15082	11475	2186	15459	44202
9.in	16919	12880	2440	17317	49556

Intra-block Quicksort	reads	writes	branches	other	Total
0.in	512	794	179	1362	2847
1.in	1216	1652	405	2982	6255
2.in	1984	2563	647	4687	9881
3.in	2756	3511	890	6431	13588
4.in	3424	4355	1107	8019	16905
5.in	4592	5626	1449	10284	21951
6.in	5120	6393	1631	11725	24869
7.in	6140	7505	1936	13757	29338
8.in	6940	8397	2186	15459	32982
9.in	7756	9437	2440	17317	36950



High Level Architecture:

For this compiler, we used a modular structure in order to incrementally output the assembly instructions. We do this by creating classes for the instruction selector and the register allocation processes. We did this in order to decrease the difficulty of debugging by being able to have a usable output at each step of the development process, as well as enabling the possibility of an extension of the program with other algorithms for instruction selection or register allocation. Within the instruction selector class, we can store information that is crucial to the register allocation class, such as a mapping of variables in the IR to virtual registers, as well as maintaining a map of the corresponding offsets from the stack pointer. Inside of the instruction selector class, we start by generating a control flow graph so that we can perform dynamic analysis on the IR stream to generate basic blocks. After we have generated the mappeed instructions using virtual registers, we then pass these instructions into a register allocator class. From here, based on the command line input, it will either run a naïve register allocator or an intra-block algorithm allocator. The naïve register allocator iterates through the instructions looking for virtual registers. When it finds one, it allocates space on the stack and records the offset from the stack pointer for that specific function. Whenever this virtual register is read or written to, it will load or store that location on the stack, respectively. In

intrablock, we use the CFG generated in the instruction selection as well as a mapping provided by the instruction selector of MIPS instruction block leaders in order to compute the live ranges for each of the virtual registers. Using these mappings, we can then assign certain virtual registers to physical registers, with any more values overflowing to the stack.

Low-level design decisions:

In instruction selection, we start by generating the CFG of the IR passed in. We go through each instruction one at a time, place it in a block and possibly place it in a set containing all of the leaders. We then map the IR instruction to the corresponding MIPS instructions. Each of these operands can add multiple instructions and have different functionality based on the location of the instruction within the CFG. All of the stack teardown and function calling functionality is added in this section, as we run through the IR file line by line, considering any special labels or function declarations. In intra-block register allocation, we start going through the instructions and we filter out all of the leader instructions for the CFG. We then go through the process of calculating the live set for every instruction within that block. We then generate the registers based on the live set of each virtual register, and the count of uses for each virtual register. Through this we are allocating the registers one block at a time, instead of the whole program. For naive register allocation, we are only ever utilizing $t8 and $t9, since the only values we need in registers are the ones that are in use in the next instruction. We began storing the virtual registers in bulk at the top of the frame for that function, but we later switched to a dynamic storage throughout the frame. We did this so we can keep a bulk of the logic outside of the assembly and have fewer instructions needed for calling and returning from functions. We keep track of this for both intra-block and naive through a nested hashmap. We have a hashmap of function label keys, with each of those having as its value a hashmap of live virtual registers and their respective offset from the current stack pointer. Because of this, we would have to increment all of the live values in the hashmap whenever we would increase the stack pointer.

Challenges:

A major challenge that arose during this project was the handling of stack setup and teardown in order to save a function's state properly before executing a function call. In order for our code to function correctly on MIPSInterpreter, we had to generate the saving and restoring of all virtual registers before and after a function call. However, when dealing with naive allocation, all virtual registers are already stored and loaded from the stack before each instruction, so additional saving of these registers was not necessary. This was similarly the case with intra-block allocation, except that temporary assigned registers which were live in a block with a function call needed to be saved and restored. As such, we had to devise a process in order to remove these generated register saves and restores when allocating registers by matching instructions to regular expressions.

Another challenge was in keeping track of the offset from the stack pointer for each virtual register. In order to deal with this, we had a map with keys of operands and values of the offset from the stack, and every time an additional virtual register was allocated on the stack, we iterated through the map and increased all the offsets by one so that they would still point to the correct spot on the stack after the stack pointer moved.

Build and Usage:

To run our compiler against an input file and generate the assembly code, execute the command ‘./run.sh {in} {out} <allocation_option>’. Here, {in} and {out} denote the input IR file and the filename of the output assembly file, respectively. For the option to specify which allocation mode to use, in place of <allocation_option> specify either ‘--naive’ or ‘--intrablock’. If no allocation mode option is provided, or if an unrecognized one is passed, then the compiler will default to using naive register allocation.