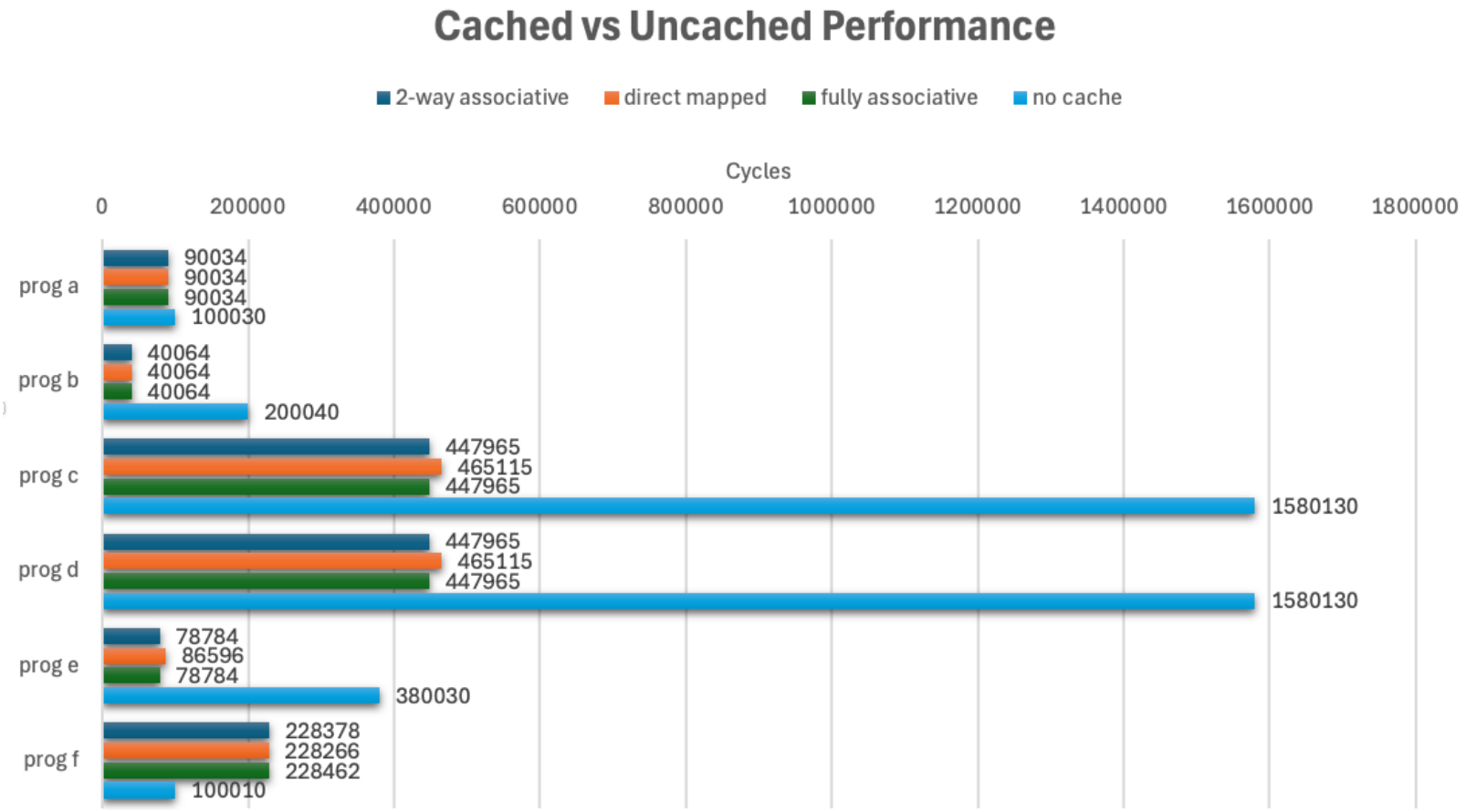
Configuration 1:

Block Size	16 bytes
Cache Lines	64 lines
Memory Size	131,072



Program A was enhanced by enabling a cache. The program consisted of 10,003 lines, 10,000 of which were subtraction operations, and the remaining were two movi and one trp. Execution was sped up by 9,996 cycles by implementing *any* kind of cache. The base, no cache value of 100,030 makes perfect sense. Reading a single word has a cost of 8 cycles, but a sequential read will cost an additional 2 cycles. Fetch implements this logic, and the result is that each fetched instruction has a cost of 10 cycles when running with the cache turned off. With a cache enabled, two instructions are pulled into the cache at once with a 16 byte block. Each miss has a 15 cycle miss penalty, with a 1 cycle hit afterwards. All misses are mandatory with this program, because the PC is constantly

incrementing. The cycle count between all cache types is identical because the whole program fits inside the cache.

Program B was also enhanced by enabling a cache. The uncached behavior of program b makes sense; there are 10,000 iterations of instructions sub and bnz. These each take 10 cycles to complete, for a total of (2 * 10) * 10,000 = 200,000 cycles. The additional 40 cycles come from the other 4 instructions in the program, each requiring 10 cycles to complete. The cached numbers also make sense; the program should repeatedly hit the cache for 2 cycles after its warm-up period. There are no conflict misses during the loop, because the two instructions fit inside the same block. The cycle count between all cache types is identical because the whole program fits inside the cache.

Program C was greatly enhanced by enabling a cache; a cache present only took 28.4% of the time that running without a cache would've taken. Program C has a loop moving through 15 instructions 10,000 times. Without a cache, each iteration through the loop costs 15 instructions * 10 cycles/fetch with an additional byte read of 8 cycles, for a total of 158 cycles per iteration, for a total of 1,580,000 cycles just from the loop. This lines up nicely with the reported number of 1,580,130, with the remaining cycles coming from the other instructions in the program. Direct-mapped has a higher cost because there are conflict misses more often, but fully associative and 2-way associative almost never kick out of the cache early, with 2-way associative pulling ahead.

Program D was also greatly enhanced by enabling a cache. In fact, program D and program C differ by exactly one instruction, ILDR vs ILDB. As such, one would expect that they would have the same cycle count as one another. As the diagram shows, they do have the exact same values, and the same analysis applies here.

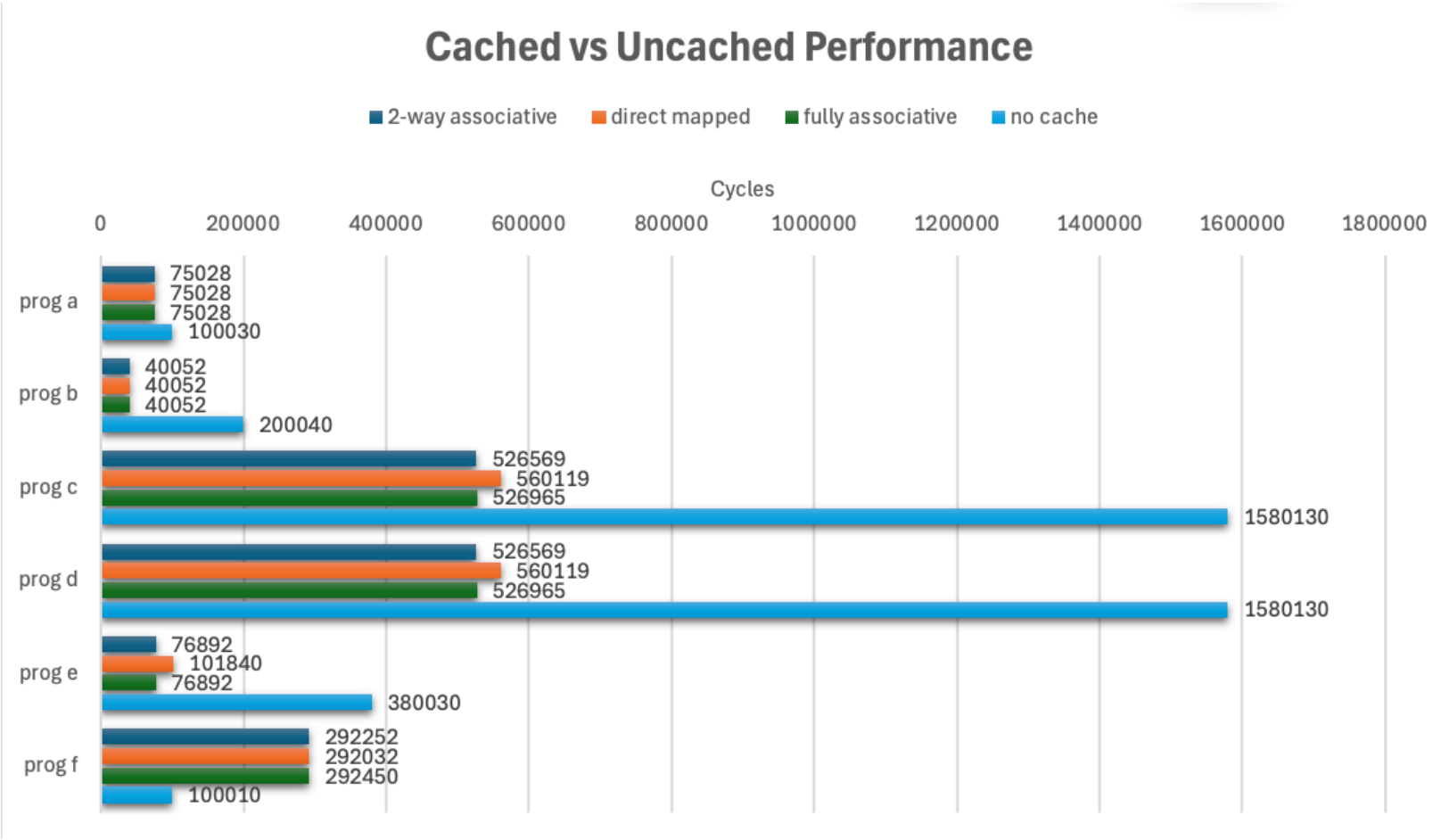
Program E is another example of a program greatly benefiting from a cache. The large uncached cycle count is amplified by the loop containing a load byte instruction. This is another example of the direct-mapped cache having conflict issues, causing sections of the block to be evicted early. The program is not large enough for a difference between 2-way and fully associative caches to differ.

Program F was the only program to perform *worse* when a cache was implemented. The file consists of 10,000 lines of seemingly random jumps between instructions, likely constructed as an adversarial test for the cache types. There is a large amount of thrashing going on within the cache when this program is run, causing blocks to be read into memory only to be quickly rewritten. The direct-mapped cache performed best of all cache types, perhaps "getting lucky" for a few cycles where the cache may have hit. The 2-way set associative cache performs better than fully associative because it both implements a LRU

eviction policy, and it likely had less conflict misses because it had more ways. In short, it had more chances for hits over the fully associative cache.

Configuration 2:

Block Size	32 bytes
Cache Lines	32 lines
Memory Size	131,072



Progam A's caches performed better with configuration 2 (cfg2) over configuration 1(cfg1). This is likely because the block size was increased to 32 bytes, reducing the number of compulsory misses in the program. This is kept across all cache types, meaning that they were all suffering from compulsory misses.

Program B's performance was improved by exactly 12 cycles in all cache types, also leading to the conclusion that compulsory misses were avoided because of the larger

block size. In this case, it was likely a single avoided compulsory miss that caused the cycle count to improve.

Program C's cached performance with cfg2 was better than not implementing a cache but performed worse than cfg1. The reduction in the number of cache lines means there was likely more index conflicts, leading to more compulsory misses over cfg1.

Program D's performance with was identical to Program C using cfg2, meaning that the caches performed correctly when executing ildr and ildb. The reduction in the number of cache lines means there were more compulsory misses over cfg1.

Program E's performance was worse with cfg2 when using a direct-mapped cache, but slightly better in any associative cache. The reduction in cache lines significantly hindered the direct-mapped cache in this case, and the increase in block size had a slight benefit with the associative caches.

Program F's performance was much, much worse with cfg2 over cfg1. The reduction in cache lines means that there were less chances for any kind of cache to "get lucky" with the random jumps.