1: Simulation of qsort_small.c with and without cache + optimization levels

Objective:
To simulate the execution of qsort_small.c using input_large.dat and input_small.dat as inputs. For each input, I compiled and ran the code with three levels of optimization: -O1, -O2, and -O3, and measured the execution time.

Compilation and Simulation Details:

1. **Compilation Command**
   I used the following command for cross-compilation targeting the ARM architecture:

   aarch64-linux-gnu-gcc -static -O{optimization_level} -o qsort_small qsort_small.c

2. **Execution Configuration**
   The paths were defined to the binary and input files in the armConfig.py file:

   binary = 'PATH_TO/qsort/qsort_small'
   input_file = 'PATH_TO/qsort/input_small.dat' # or 'input_large.dat' for larger input
   process.cmd = [binary, input_file]

Cache Configuration:

```
# Define L1 Cache
class L1Cache(Cache):
  assoc = 2
  tag_latency = 2
  data_latency = 2
  response_latency = 2
  mshrs = 4
  tgts_per_mshr = 20

  def connectCPU(self, cpu):
    raise NotImplementedError

  def connectBus(self, bus):
    self.mem_side = bus.cpu_side_ports


# Instruction and Data L1 Cache subclasses
class L1ICache(L1Cache):
  size = '16kB'

  def connectCPU(self, cpu):
    self.cpu_side = cpu.icache_port
```

```
class L1DCache(L1Cache):
  size = '64kB'

  def connectCPU(self, cpu):
    self.cpu_side = cpu.dcache_port


# Define L2 Cache
class L2Cache(Cache):
  size = '256kB'
  assoc = 8
  tag_latency = 20
  data_latency = 20
  response_latency = 20
  mshrs = 20
  tgts_per_mshr = 12

  def connectCPUSideBus(self, bus):
    self.cpu_side = bus.mem_side_ports

  def connectMemSideBus(self, bus):
    self.mem_side = bus.cpu_side_ports
```

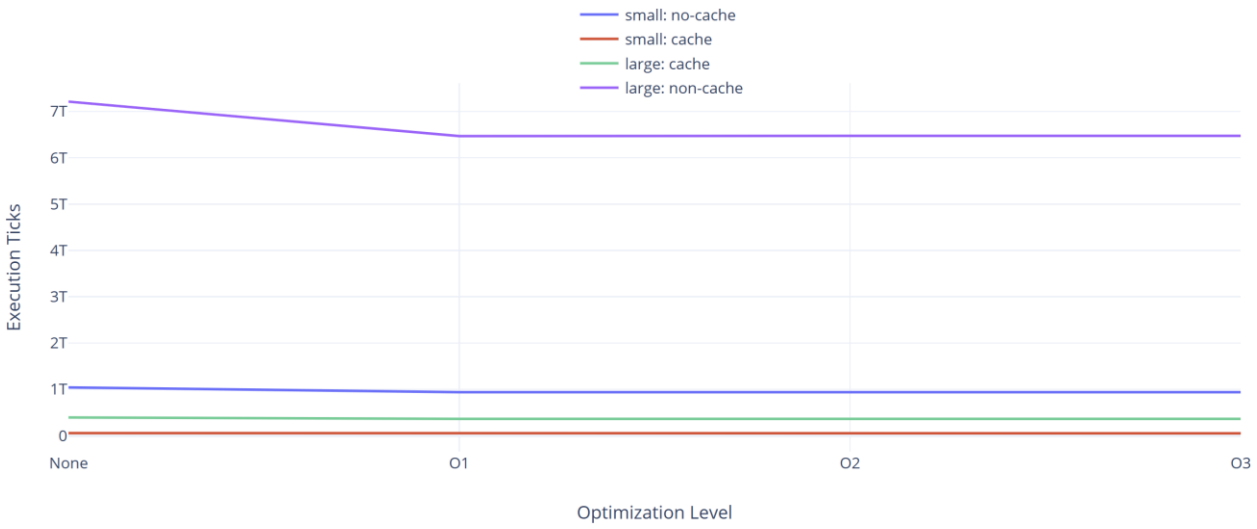
Significance: Cache memory plays a crucial role in modern processors by providing quick access to frequently used data, minimizing the need to fetch data from the slower main memory. For memory-intensive applications like sorting large datasets, cache can significantly reduce execution time by reducing memory latency, especially in cases where data reuse is high.

Results [Cached & Non-Cached Configurations]:

Optimization Level	Input File	Execution Ticks (Non-Cached)	Execution Ticks (Cached)
None	input_small.dat	1040572066000	55233708000
O1	input_small.dat	938433404000	49855390000
O2	input_small.dat	939312935000	49880083000
O3	input_small.dat	939312935000	49880083000
None	input_large.dat	7211287711000	391129337000
O1	input_large.dat	6468217399000	362417677000
O2	input_large.dat	6473329441000	362462142000
O3	input_large.dat	6473329441000	362462142000

Note: *Data is recorded in execution ticks, which are small time units in gem5 simulator used to track instruction execution with high precision. To get execution time, you'll need to divide ticks by* 10^{12}*. Although the execution times are relatively short, the tick values appear large due to this precision.*

Visual Plot:



Observations:

1. **Impact of Optimization Levels (-O1, -O2, and -O3):**
   - **-O1:** At the first optimization level, the compiler introduces basic optimizations that generally reduce code size and remove redundant calculations. For qsort_small.c,

this optimization level provides a noticeable improvement in execution ticks compared to the non-optimized version, with reductions in both cached and non-cached configurations. However, the effect diminishes as higher levels of optimization are introduced, indicating that the code benefits from initial but limited performance gains from optimization.

- **-O2:** At the second level, the compiler applies more aggressive optimizations, like loop unrolling and additional inlining, which aim to improve computational speed. However, in this case, the execution ticks show minimal improvement compared to -O1, suggesting that these additional optimizations do not significantly benefit the specific workload or data access patterns in qsort_small.c.
- **-O3:** The third level introduces the highest degree of optimization, focusing on maximizing speed through aggressive code restructuring. While -O3 generally leads to faster code in more computation-intensive programs, it does not lead to additional improvements over -O2 for qsort_small.c. This consistency in execution ticks across -O2 and -O3 implies that the workload may not benefit from further computational optimizations or that memory access patterns dominate its runtime.

2. **Effect of Caching:**
   - Caching greatly enhances performance for both input sizes. The cached configuration drastically reduces execution ticks compared to the non-cached setup, underscoring the importance of cache for workloads with high memory access demands.
   - **For input_small.dat:** The execution ticks drop from 1.04 trillion (non-cached) to 55.2 billion (cached) without optimizations, which is nearly a 19x improvement. This difference highlights how caching reduces memory access time by holding frequently used data closer to the processor.
   - **For input_large.dat:** The impact is even more pronounced, with non-cached execution at 7.21 trillion ticks compared to 391.1 billion ticks in the cached scenario (without optimizations), achieving nearly an 18x speedup. Larger datasets benefit more significantly due to the larger volume of repeated memory accesses that caching optimizes.
3. **Comparison by Input Size:**
   - **Smaller Input (input_small.dat):** This file requires fewer memory accesses due to its size, so while caching improves performance, the relative improvement is smaller than for larger datasets.
   - **Larger Input (input_large.dat):** This dataset incurs substantially longer execution times, with a much larger volume of memory accesses, especially in the non-cached configuration. The larger data size results in greater benefit from caching, as the cache is able to store and quickly retrieve frequently accessed data, reducing latency and overall execution time more dramatically than for the smaller dataset.

Conclusion:

This analysis underscores the critical role of caching over compiler optimization for memory-intensive applications like sorting large datasets. While compiler optimizations at levels -O1, -O2, and -O3 yield some performance benefits, they have a far smaller impact on runtime than caching does in this case. The results suggest that caching is indispensable for applications that repeatedly access large datasets, as it provides a significant reduction in memory latency by holding frequently used data close to the processor. For this workload, caching provides close to an order of magnitude improvement in execution time, making it the dominant factor in optimizing performance.

Overall, while compiler optimizations refine code execution efficiency, they are less impactful than caching for memory-bound tasks.

2: Simulation of basicmath_small.c with and without cache

Objective:
I ran basicmath_small.c (without print statements) on three different processor types - ArmTimingSimpleCPU, ArmO3CPU, and ArmMinorCPU - both with and without the cache configuration, at different optimization levels to measure and compare execution times.

Processor Types and Configurations:

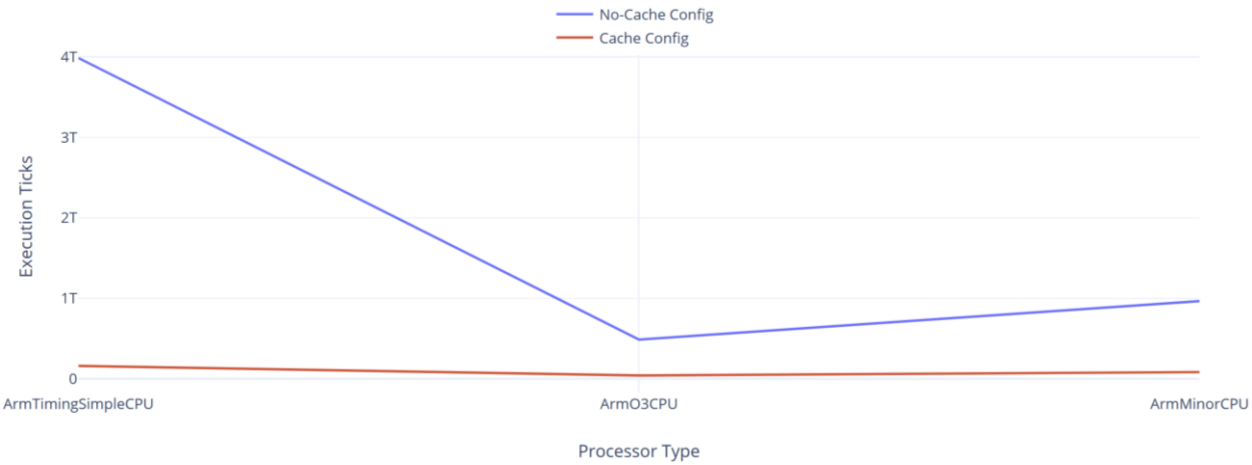
1. **Processor Types:**
    - ArmTimingSimpleCPU
    - ArmO3CPU
    - ArmMinorCPU
2. **Cache Configurations:**
    - With and without the configured cache (specified in the first task)

Results [Optimization + Cached & Non-Cached Configurations]:

Processor Type	Optimization Level	Execution Ticks (Non-Cached)	Execution Ticks (Cached)
ArmTimingSimpleCPU	None	3984589649000	160899681000
ArmTimingSimpleCPU	O1	3804748021000	77420708000
ArmTimingSimpleCPU	O2	3795103280000	76880392000
ArmTimingSimpleCPU	O3	3795103280000	76880392000
ArmO3CPU	None	487873756000	42514429000
ArmO3CPU	O1	485118091000	41840371000
ArmO3CPU	O2	484992100000	41750029000
ArmO3CPU	O3	484992100000	41750029000
ArmMinorCPU	None	965174606000	83675978000
ArmMinorCPU	O1	955621049000	83054873000
ArmMinorCPU	O2	954003301000	82845706000
ArmMinorCPU	O3	954003301000	82845706000

Note: *Data is recorded in execution ticks, which are small time units in gem5 simulator used to track instruction execution with high precision. To get execution time, you'll need to divide ticks by* 10^{12}*. Although the execution times are relatively short, the tick values appear large due to this precision.*

Visual Plot [Example of Non-Optimized Runs]:



Observations:

1. **CPU Architecture Performance Comparison:**
    - **ArmO3CPU** demonstrates the best performance across all configurations, both cached and non-cached, due to its out-of-order execution capabilities. This CPU architecture is designed to handle multiple instructions at once, reducing idle time and efficiently utilizing the processor.
    - **ArmTimingSimpleCPU** has the longest execution time in all configurations due to its in-order execution design. Without the cache, its execution ticks are considerably higher, as each memory access incurs a delay. This CPU type benefits the most from caching, which helps mitigate the delays caused by frequent memory access.
    - **ArmMinorCPU** falls between ArmO3CPU and ArmTimingSimpleCPU in terms of performance. Its semi-sophisticated architecture provides better handling of instructions than ArmTimingSimpleCPU but still lags behind the fully out-of-order design of ArmO3CPU.
2. **Impact of Cache:**
    - **Cache Effectiveness Across Architectures:** Caching drastically reduces execution ticks for all processors, highlighting its importance in memory-intensive tasks like basicmath_small.c. The reduction is most pronounced in the ArmTimingSimpleCPU, where caching brings the execution ticks down from around 3.98 trillion to 160.9 billion (unoptimized). This demonstrates how caching alleviates memory access delays, especially for in-order CPUs that cannot hide these latencies as effectively as out-of-order designs.
    - **Cached vs. Non-Cached Trends:** For ArmO3CPU and ArmMinorCPU, the cached configuration provides significant performance boosts, though the relative improvement is less dramatic than for ArmTimingSimpleCPU. This is because out-

of-order and semi-out-of-order designs can partially hide memory latencies even without a cache, though caching still enhances overall efficiency.

3. **Impact of Optimization Levels (-O1, -O2, -O3):**
    - **Minimal Gains Beyond -O1:** Across all CPU types, the optimization levels show diminishing returns. The most substantial drop in execution ticks occurs between the unoptimized version and the -O1 level. However, from -O1 to -O3, the performance improvement is minimal, especially in the cached configurations. This suggests that the basic optimizations at -O1 are sufficient to reduce redundant computations, while higher levels do not significantly benefit this workload, which is likely constrained more by memory access patterns than by computational complexity.

Conclusion:

These results underscore the dual importance of CPU architecture and caching in achieving optimal performance for memory-intensive tasks. The ArmO3CPU, with its out-of-order execution design, delivers the fastest performance, particularly when combined with caching. This combination effectively reduces memory access latencies while also managing instructions efficiently. On the other hand, simpler CPUs like ArmTimingSimpleCPU are highly reliant on caching to improve performance, as they are more susceptible to delays from memory accesses. Optimization levels beyond -O1 provide limited gains, indicating that basic optimizations are sufficient for this workload. For tasks involving frequent memory access, an advanced CPU architecture with effective caching yields the most substantial performance benefits.

3: Rewriting basicmath_small.c in ARM RISC Assembly

To analyze performance differences between C code and ARM assembly, I rewrote basicmath_small.c in ARM RISC Assembly. This allowed for a direct comparison of how each language and optimization strategy influences execution times, especially given the low-level nature of assembly, which provides more control over CPU instructions and memory access.

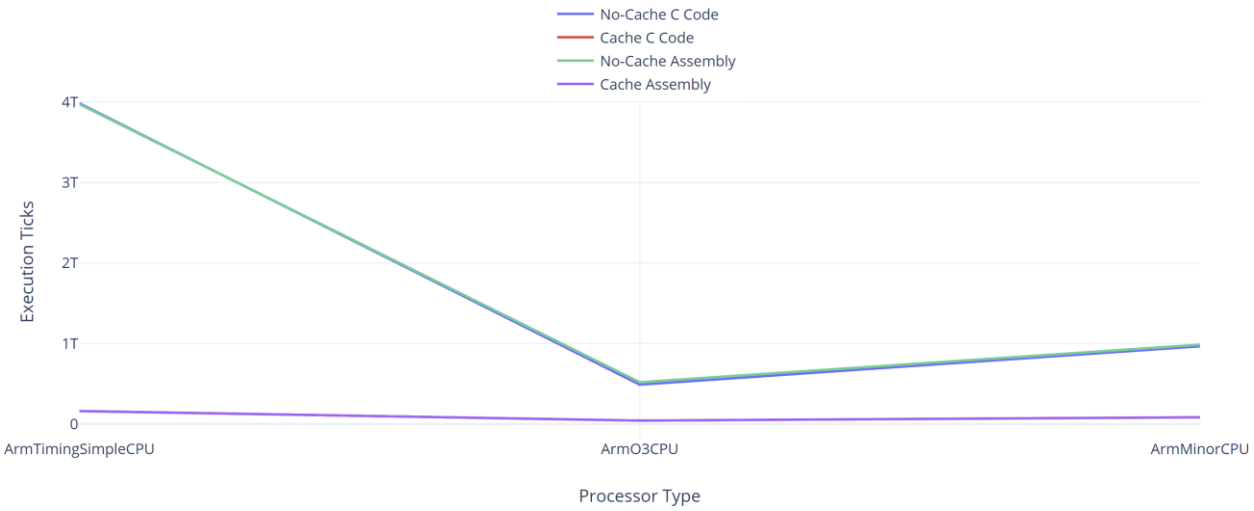
After translating the logic of basicmath_small.c to ARM assembly, I re-ran all configurations with the new assembly code. This included testing with three different CPU types (ArmTimingSimpleCPU, ArmO3CPU, and ArmMinorCPU) and both cached and non-cached configurations to determine the impact of assembly-level optimizations on execution time.

Results Comparison: C Code vs. ARM Assembly

Processor Type	Exec Ticks (C Code) (Non-cached)	Exec Ticks (Assembly) (Non-cached)	Exec Ticks (C Code) (Cached)	Exec Ticks (Assembly) (Cached)
ArmTimingSimpleCPU	3984589649000	3973000502000	160899681000	160433826000
ArmTimingSimpleCPU:O1	3804748021000	3798223380000	77420708000	771322812000
ArmTimingSimpleCPU:O2	3795103280000	3782330112000	76880392000	762277191000
ArmTimingSimpleCPU:O3	3795103280000	3782330112000	76880392000	762277191000
ArmO3CPU	487873756000	506120086000	42514429000	42255636000
ArmO3CPU:O1	485118091000	501122732000	41840371000	41512238000
ArmO3CPU:O2	484992100000	499912203000	41750029000	41255407000
ArmO3CPU:O3	484992100000	499912203000	41750029000	41255407000
ArmMinorCPU	965174606000	972747502000	83675978000	80639224000
ArmMinorCPU:O1	955621049000	961443281000	83054873000	79998229000
ArmMinorCPU:O2	954003301000	959911321000	82845706000	79401140000
ArmMinorCPU:O3	954003301000	959911321000	82845706000	79401140000

Note: These tests were conducted with print statements excluded

Visual Plot [Example of Non-Optimized Runs]:



Observations:

1. **ArmTimingSimpleCPU:**
    - The non-cached assembly version shows a slight improvement over the C code, which can be attributed to the reduced instruction overhead and more efficient memory handling that assembly allows. In this architecture, the in-order execution makes assembly optimizations beneficial, as they streamline memory accesses and reduce unnecessary instructions.
    - With caching enabled, the difference between C and assembly becomes minimal. This indicates that caching mitigates the need for fine-tuned memory access management, as both C and assembly perform similarly under cache support. This suggests that, for simpler CPU architectures, the primary bottleneck is memory access rather than instruction efficiency.
2. **ArmO3CPU:**
    - The non-cached C code performs slightly better than the assembly version. ArmO3CPU's out-of-order execution architecture allows it to efficiently reorder instructions for optimal performance, so compiler-generated code at higher optimization levels (-O2 and -O3) can take full advantage of this. In this case, manually optimized assembly code sees little to no performance gain over the compiler's optimizations.
    - With caching, both C and assembly perform nearly the same. This implies that caching reduces memory latencies, which in turn diminishes the potential gains from hand-tuned assembly. In a high-performance CPU like ArmO3CPU, the compiler's optimizations and out-of-order capabilities are generally sufficient to achieve efficient execution times without the need for manual instruction control.

3. **ArmMinorCPU:**
    - Without cache, the execution time of assembly is close to that of the C code. ArmMinorCPU's pipelined in-order design benefits modestly from assembly-level optimizations, but since it cannot reorder instructions, the gains from assembly are limited.
    - With caching, the assembly code is marginally faster than the C code, suggesting that the efficient register usage in assembly code can slightly enhance performance by reducing memory accesses. However, the benefit is minor, as the CPU's simpler architecture does not leverage complex optimizations like those seen in out-of-order processors.

Conclusion:

These results illustrate the varying impact of assembly-level optimizations across CPU architectures and cache configurations. ArmO3CPU, with its advanced out-of-order execution and effective compiler optimizations, benefits the least from manual assembly code, as the CPU's design and caching effectively handle performance bottlenecks. In contrast, ArmTimingSimpleCPU and ArmMinorCPU show minor gains with assembly, especially in non-cached configurations, due to the finer control over memory access and instruction flow that assembly provides.

Overall, caching significantly reduces the performance gap between C and assembly for all processors, highlighting that cache memory can often achieve or surpass the benefits of manual assembly coding. This underscores that in modern CPU architectures, compiler optimizations and cache effectiveness can make high-level languages like C nearly as performant as assembly, simplifying development without major sacrifices in execution speed.

Summary of Observed Trends

General Trends Observed:

- Across all tasks, cache configurations had a consistently large impact on execution time, especially for larger inputs and simpler CPU types.
- CPU architecture significantly affects performance: ArmO3CPU consistently provided the best performance, indicating the benefits of out-of-order execution for memory-intensive tasks.
- Assembly-level optimizations had a greater impact on simpler CPUs (ArmTimingSimpleCPU), whereas advanced CPUs (ArmO3CPU) saw minimal differences between C and Assembly, as they could rely more on compiler and CPU-level optimizations.
- Overall, the combination of caching and an advanced CPU architecture (ArmO3CPU) produced the best performance results, underscoring the importance of both memory management and CPU design for optimized performance.