To run the experiment, we have used a bash script to generate all the necessary simulation output. And then sorted Protocol based on Cycles in ascending order. We have found that MOESIF outperforms all the other protocols in all benchmarks. In the following sections for each source, we will present our simulated analysis and reasoning.

1. Array Sum

(a) Array Stripe

Protocol	Trace Directory	Cycles	Cache Misses	Cache Accesses	$-to-$ Transfers	Memory Reads	Memory Writes
MOESIF_PRO	experiments/traces/array_stripe/core_8	6680	287	2760	251	36	7
MESI_PRO	experiments/traces/array_stripe/core_8	26196	287	2760	42	245	7
MSI_PRO	experiments/traces/array_stripe/core_8	28751	287	2760	7	280	7
MOSI_PRO	experiments/traces/array_stripe/core_8	28751	287	2760	7	280	7

MOESIF won in terms of Cycles. It's because it requires very few Memory reads which is very expensive and has a very large number of cycles as a penalty. In the source code of array_sum, the value of an array element is being summed. So multiple processors will concurrently access and modify shared data(sum). And MOESIF performs better due to its optimizations in minimizing bus traffic, efficient handling of shared data

(b) Array Block

MOESIF_PRO	experiments/traces/array_block/core_8	4516	65	2736	29	36	7
MESI_PRO	experiments/traces/array_block/core_8	5493	65	2736	17	48	7
MSI_PRO	experiments/traces/array_block/core_8	6340	65	2736	7	58	7
MOSI_PRO	experiments/traces/array_block/core_8	6340	65	2736	7	58	7

MOESIF has the lowest cycles for Array block also.

Why does performance differ in array_sripe and array_block?
Ans: Because array_block source has more spatial locality and better cache accesses. No processor is competing for the same block in array_block, unlike array_stripe source code. This is why implementation in array_block has better runtime.

2. Matrix Multiply

MOESIF performs better due to its optimizations in minimizing bus traffic, efficient handling of shared data, and ownership management. These features are particularly advantageous in scenarios where multiple processors are concurrently accessing and modifying shared data, as is the case in matrix multiplication.

(a) MatMul Rows

MOESIF_PRO	experiments/traces/matmul_row/core_8	54322	848	92400	759	101	420
MESI_PRO	experiments/traces/matmul_row/core_8	58977	790	92400	574	230	540
MSI_PRO	experiments/traces/matmul_row/core_8	59906	810	92400	553	265	553
MOSI_PRO	experiments/traces/matmul_row/core_8	62913	724	92400	606	264	327

(b) MatMul Cols

MOESIF_PRO	experiments/traces/matmul_col/core_8	121722	1974	118504	2062	101	1338
MOSI_PRO	experiments/traces/matmul_col/core_8	170598	2565	118504	3505	359	1824
MESI_PRO	experiments/traces/matmul_col/core_8	195788	2800	118504	2358	751	2320
MSI_PRO	experiments/traces/matmul_col/core_8	227346	3302	118504	2744	999	2744

Why does performance differ in matmul_rows and matmul_col?

Ans: In matmul_rows we observe better performance. It's because of spatial locality and cache-friendly access pattern. In matmul_rows , source code can leverage accessing contiguous memory locations within each thread's computation.

3. Pipeline

In this code, multiple threads are used to perform parallel computation of partial products for a set of values. Each thread takes work items from one stage, performs computation on them, and pushes them to the next stage. The final stage contains the computed partial products, which are then compared with pre-calculated values to check correctness. MOESIF could minimize bus transactions and reduce latency, potentially resulting in better performance compared to other protocols.

MOESIF_PRO	experiments/traces/pipeline/core_8	80351	846	9483	1343	54	772
MOSI_PRO	experiments/traces/pipeline/core_8	84559	818	9483	1378	135	726
MESI_PRO	experiments/traces/pipeline/core_8	107552	828	9483	827	556	813
MSI_PRO	experiments/traces/pipeline/core_8	108778	836	9483	828	567	828

4. Map-Reduce

MOESIF_PRO	experiments/traces/mapreduce/core_8	53324	679	11281	986	52	512
MOSI_PRO	experiments/traces/mapreduce/core_8	61319	620	11281	879	151	462
MESI_PRO	experiments/traces/mapreduce/core_8	72421	636	11281	579	414	562
MSI_PRO	experiments/traces/mapreduce/core_8	73327	637	11281	570	424	570

In map-reduce, multiple processors are used to perform parallel computation of summing up elements of an array. Each proc is responsible for generating random numbers and pushing them into a queue associated with its respective processor. Then, the proc collectively pop elements from their own queues, perform local summation and finally perform a reduction to obtain the global sum. Since multiple proc are accessing and modifying shared queues concurrently, MOESIF is likely to perform better.