Design and Evaluation of a MESI-Coherent Multiprocessor Simulator for Parallel Dot Product

Daniel Cob Beirute
email: danielcob@estudiantec.cr
Escuela de Ingeniería en Computadores
Instituto Tecnológico de Costa Rica

***Abstract*—This work presents the design and implementation of a multiprocessor system model that enforces cache coherence using the MESI protocol. The objective is to analyze the functional behavior and performance of the architecture in a parallel dot product application with two double-precision floating-point vectors. The proposed system consists of four processing elements (PEs), each with a private cache connected through a shared bus that serializes all coherence and memory transactions. The cache subsystem implements a 2-way set associative design with write-back and write-allocate policies, managed by a least recently used (LRU) replacement policy. A segmented memory structure allows efficient data distribution and synchronization across PEs. Simulation results confirm the functional correctness of the model, demonstrating high cache hit ratios and low invalidation traffic, validating the effectiveness of MESI coherence in maintaining data consistency. Furthermore, experiments with data misalignment reveal a clear degradation in performance, manifested as increased bus traffic and cache misses, illustrating the sensitivity of shared-memory systems to alignment and locality. These findings reinforce the importance of coherent cache hierarchies and data organization in achieving high-performance multiprocessor systems.**

***Palabras clave*—Multiprocessor systems, cache coherence, MESI protocol, shared memory, bus arbitration, memory consistency, parallel computing, data alignment, performance analysis**

I. Introduction

Modern computer systems rely on multiprocessor architectures that require efficient and consistent cache management to maximize the performance of applications that use thread- and data-level parallelism techniques. The MESI protocol [1] is one of the best known for cache coherence.

In this paper we will design and implement a MESI-coherent multiprocessor model, to analyze how the system behaves and responds in a parallel dot product application with two double precision floating-point vectors.

The first part of the document will focus on the proposed design, and a further implementation analysis will explain the fine details of the model. After that, the model results in terms of behavior and performance will be revised against several vector conditions. At last, conclusions and recommendations for further implementations will be exposed at the end.

II. Design and Implementation

The proposed multiprocessor model is a four processing element (PE) system with private cache for each PE. The cache will make requests to the bus, that acts as a interconnect module and will handle and serialize all requests. If necessary the bus will make a request to the memory so it can retrieve or store data. This model can be viewed in fig. 1. In the simulation the hardware components will execute in separate threads to better model the parallel activities that occur in a multiprocessor system.

Processing a parallel dot product operation with two vectors of size N will require the PEs to distribute vector data. Each PE will process N/4 elements of the vector, and a master PE will apply the final reduction of the operation based on the results of the other PEs. In case the vector is not a multiple of four, the residue will be processed by the master PE.

This workflow requires a segmented memory divided between a data section and a shared memory section. The data section contains the two vectors that will be processed by the PEs. On the other hand the shared memory section contains relevant constants required to process the vector and parallelize the workflow; data like the starting index, segment size, and result address of each PE, vector addresses, flag address (required for the synchronization of the final reduction) and the final result address will be contained here. With a segmented memory like this the PEs can execute the same assembly code without caring for the size of the vectors.

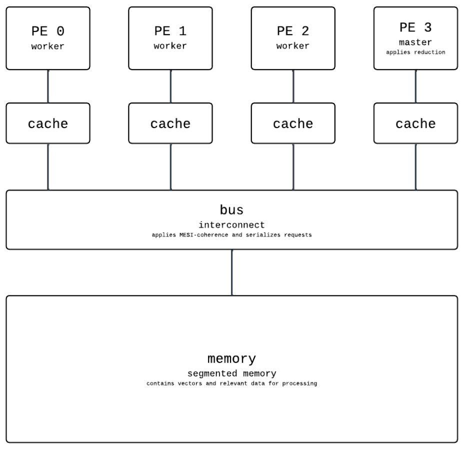


Fig. 1. Proposed Multiprocessor Model.

The implementation of the design requires a more clear definition of the methods, protocols and policies, each component of the system will apply. In the next subsections will go into the details of each component.

A. Processing Element Instruction Set

Each PE contains 8 registers of 8 bytes and share the same set of instructions, all capable of managing double-precision floating-point operations. Fig. 2 shows the complete set of instructions.

Instruction	Operands	Example
LOAD	R<dest>, R[<addr>]	LOAD R0, [R1]
STORE	R<dest>, R[<addr>]	STORE R0, [R1]
MOV	R<dest>, <num>	MOV R0, 4.3
FMUL	R<dest>, R<op1>, R<op2>	FMUL R0, R1, R2
FADD	R<dest>, R<op1>, R<op2>	FADD R0, R1, R2
INC	R<dest>	INC R0
DEC	R<dest>	DEC R0
JNZ	<label>	JNZ LOOP
HALT	NONE	HALT

Fig. 2. Implemented Instruction Set.

The `LOAD` instruction searches the memory for the specified address and stores the data in the destiny register. On the other hand, the `STORE` instruction writes in the specified memory address the data contained in the destiny register. This two functions are the only ones that interact with the cache component. Even though the instruction states they access memory, this is not exactly true. Each PE can only interact and communicate with their own cache component. The cache will communicate with the bus if necessary to load and store data, but this will be specified in the cache section.

The `MOV` instruction loads an immediate value to the destiny register. This is useful in the dot product program, as we can load known memory values like the address of each vector. The `FMUL` instruction executes the floating point version of the multiplication operation. The result is stored in the specified destiny register. The `op1` and `op2` registers specify the values that will be multiplied. The `FADD` instruction executes the floating point version of the sum operation. The result is stored in the specified destiny register. The `op1` and `op2` registers specify the values that will be sum together.

The `INC` instruction executes the `i++` operation to the value specified in the destiny register, and will store the result in the same register. The `DEC` instruction executes the `i--` operation to the value specified in the destiny register, and will store the result in the same register. The `JPZ` instruction will execute a PC jump to the specified label if the previous operation raised the zero flag. If the flag is not raised the program will continue the execution normally to the next PC value. The `HALT` instruction will tell the PE to stop the execution of the program. After halting the program the PE will tell it's own cache to write-back all the cache lines with the modified state.

It is important to note that the PE is the only hardware component not modeled as a separate thread. This is due to the fact that the proposed system does not implement out of order execution. Without this, the PE needs to stop execution until the cache contains the necessary data to complete the instruction. Therefor, the implemented PE will execute in the same thread as the cache component.

B. Cache

The implemented cache is 2-way set associative with 16 lines of 32 bytes. It implements a write-back and write-allocate policy, and a least recently used (LRU) policy for eviction. In fig. 3, the general structure of a cache line can be seen. The MESI state of the block is stored in the state segment of the line. The calculations of the tag and set of a specific memory address are done following the conventions presented in [1]

Consider a block of data as 4 words of 8 bytes. When a memory address is requested by the PE, the cache calculates the offset of the address based on its nearer aligned block (rounded down). The cache then calculates the corresponding set assigned to the aligned memory address, and searches both ways for its tag. If the tag is found and is valid, the requested word (according to the calculated offset) is returned to the PE. It the block is not found, the cache requests the bus for it's address, and stores the entire block in a available way. This gives the cache spatial locality, reducing, in most use cases, the cache miss percentage.

If both ways are occupied, the cache evicts the least recently used block, according to the LRU bit. LRU policy also helps in reducing the cache miss percentage, as it gives temporal locality to the lines in cache. To apply the LRU policy, when a cache line is accessed by the PE the LRU bit of that way is set to 1. Then, the LRU bit of the other way corresponding to the same set is set to 0.

Depending of the operation done by the PE (read/write), and the MESI state of the line; the cache makes the according request to the bus. If the PE requests a read operation for a block that has not been allocated, the cache makes a `BUS_RD` request. If the PE requests a write operation for a block that has not been allocated, the cache makes a `BUS_RDX` request, allocates the block, and writes the corresponding word (write-allocate policy). If the PE writes on a block with a shared MESI state, the cache makes a `BUS_UPGR` request. If the cache evicts a block with a modified MESI state, the cache makes a `BUS_WB` request (write-back policy).

In the simulation the cache component contains a mutex, in order for the bus to handle requests without risking a deadlock or race condition.

tag	state	lru
data		valid

Fig. 3. Structure of the implemented cache line.

C. Bus

The bus maintains cache coherence across all the system and is the only component in direct contact with the memory. As a result of that the bus is the most important component for the system to function properly. As all PEs and caches execute in parallel, multiple requests can be made at the same moment. To solve this problem the bus schedules the requests and resolves them in a serial manner. The bus implements the Round-Robin schedule algorithm, where the bus resolves a request in order based on a circular list containing all PEs. By design each PE can only make one request at a time (out of order execution is not implemented), so the depth of each queue (one per PE) has a maximum size of one.

There are four types of requests a cache can send to the bus. According to the type, the bus must make a series of steps. In fig. 4, a graph of the MESI states and their transitions can be used as a guide while reading this section. Keep in mind, all cache lines start with the invalid state.

Starting with `BUS_RD` the bus checks if there are other caches with the requested block stored. If the search is a hit in another cache, the bus must send the data back to the requester, and set the shared state to both lines. In this step, if the state of the hit line is modified, the bus must execute a write-back to memory. If the search is a miss in each cache, then the bus accesses the block from memory and brings it back to the requester, setting the state to exclusive.

If a cache sends `BUS_RDX`, the bus checks if there are other caches with the requested block stored. If the search is a hit in another cache, the bus must send the data back to the requester, and set the modified state to the requester line and invalid state to each cache with the line. In this step if the state of the hit line is modified, the bus makes a write-back to memory. If the search is a miss in each cache, then the bus accesses the block from memory and brings it back to the requester, where the PE writes the data, and sets the state to modified.

If a cache sends `BUS_UPGR`, the bus must check if other caches have the line and set the state to invalid. After that the bus must set the requester cache line to modified.

If a cache sends `BUS_WB`, the bus must execute a write-back to memory at the specified address.

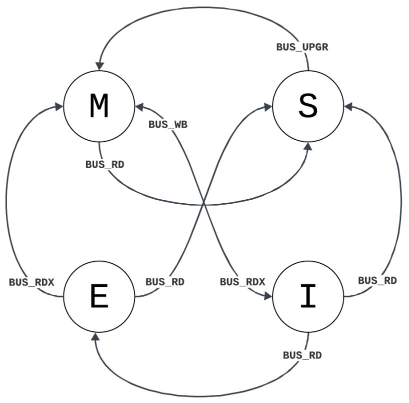


Fig. 4. MESI states with transitions, viewed from the perspective of the cache making the request.

D. Memory

The memory contains 512 positions (addresses), each containing 8 bytes of data. Meaning the memory can hold up to 128 blocks. For the dot product application the memory is distributed as explained at the start of the section. The memory contains a method to load vectors with a misalignment offset, in order to experiment with the consequences of misaligned data.

The memory component contains a mutex, in order for the bus to make read and write operations sequentially.

III. Results and Analysis

The validation phase confirmed the correct functional behavior of the multiprocessor model. Figure 5 shows the expected and computed outputs for a dot product operation with two 16-element vectors. The calculated result matches the theoretical expectation, demonstrating that the instruction set, cache interactions, and bus transactions operate as intended. The synchronization mechanism between the PEs and the master reduction process also completed successfully, validating the correctness of the shared-memory coordination.

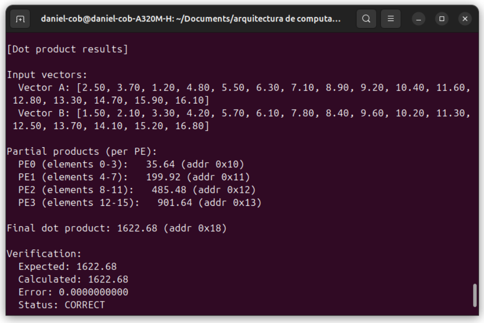
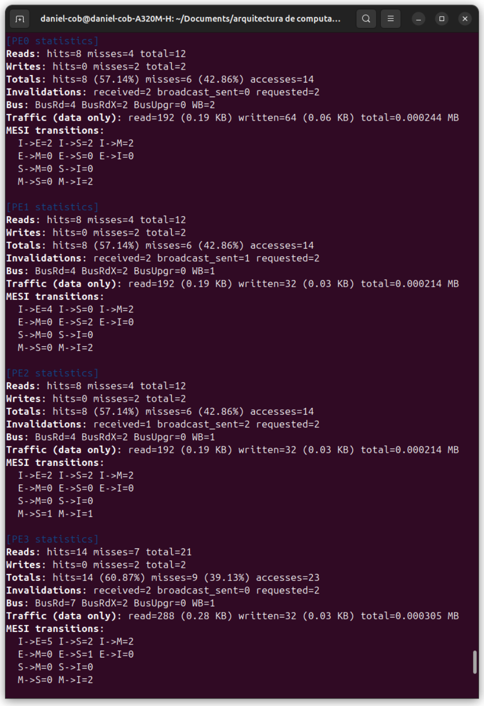

```
daniel-cob@daniel-cob-A320M-H: ~/Documents/arquitectura de computa...

[Dot product results]

Input vectors:
  Vector A: [2.50, 3.70, 1.20, 4.80, 5.50, 6.30, 7.10, 8.90, 9.20, 10.40, 11.60,
 12.80, 13.30, 14.70, 15.90, 16.10]
  Vector B: [1.50, 2.10, 3.30, 4.20, 5.70, 6.10, 7.80, 8.40, 9.60, 10.20, 11.30,
 12.50, 13.70, 14.10, 15.20, 16.80]

Partial products (per PE):
  PE0 (elements 0-3):   35.64 (addr 0x10)
  PE1 (elements 4-7):   199.92 (addr 0x11)
  PE2 (elements 8-11):   485.48 (addr 0x12)
  PE3 (elements 12-15):   901.64 (addr 0x13)

Final dot product: 1622.68 (addr 0x18)

Verification:
  Expected: 1622.68
  Calculated: 1622.68
  Error: 0.0000000000
  Status: CORRECT
```

Fig. 5. Result of the execution of a dot product program with a size 16 vector.

Each PE recorded its own statistics, summarized in Figure 6. The majority of read and write operations resulted in cache hits, validating the spatial and temporal locality advantages of the 2-way set-associative structure with LRU replacement. The bus requests made by the caches aligned with the MESI state transitions, confirming that the coherence protocol operated correctly.

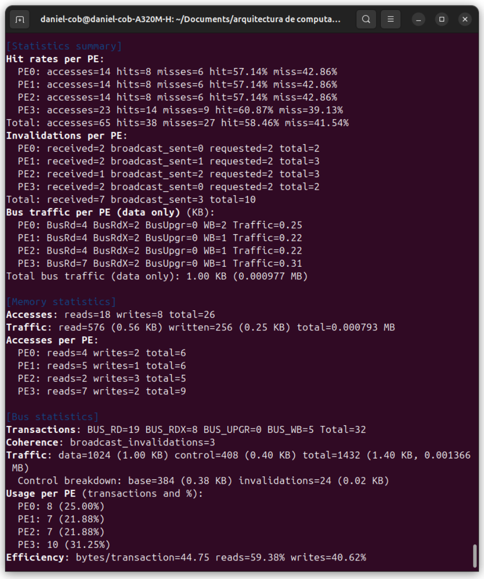
Memory statistics shown in fig. 7, show that data traffic is much smaller in contrast with the total bus traffic. Showing the efficiency of the bus reducing the total reads/writes in memory, which is slower to access in a ground-truth system. Usage per PE shows the extra computations made in the master PE when making the final reduction. The total traffic destined to invalidations is much smaller than normal control signals required to communicate the caches. This shows that the effectiveness of the system during the communication of caches.

The same program was executed with the same vectors, but with a misalignment offset of 3. The results show a increase in the read/write operations done by the cache, as well a increase in cache requests to the bus. Data misalignment, results in a increase in the total traffic across the whole system. In a real computer with latencies between memories, this can result in a significant decrease in performance. This happens because misalignment interferes with the optimizations done in cache with spatial and temporal locality.


daniel-cob@daniel-cob-A320M-H: ~/Documents/arquitectura de computa...

```
[PE0 statistics]
Reads: hits=8 misses=4 total=12
Writes: hits=0 misses=2 total=2
Totals: hits=8 (57.14%) misses=6 (42.86%) accesses=14
Invalidations: received=2 broadcast_sent=0 requested=2
Bus: BusRd=4 BusRdX=2 BusUpgr=0 WB=2
Traffic (data only): read=192 (0.19 KB) written=64 (0.06 KB) total=0.000244 MB
MESI transitions:
  I->E=2 I->S=2 I->M=2
  E->M=0 E->S=0 E->I=0
  S->M=0 S->I=0
  M->S=0 M->I=2

[PE1 statistics]
Reads: hits=8 misses=4 total=12
Writes: hits=0 misses=2 total=2
Totals: hits=8 (57.14%) misses=6 (42.86%) accesses=14
Invalidations: received=2 broadcast_sent=1 requested=2
Bus: BusRd=4 BusRdX=2 BusUpgr=0 WB=1
Traffic (data only): read=192 (0.19 KB) written=32 (0.03 KB) total=0.000214 MB
MESI transitions:
  I->E=4 I->S=0 I->M=2
  E->M=0 E->S=2 E->I=0
  S->M=0 S->I=0
  M->S=0 M->I=2

[PE2 statistics]
Reads: hits=8 misses=4 total=12
Writes: hits=0 misses=2 total=2
Totals: hits=8 (57.14%) misses=6 (42.86%) accesses=14
Invalidations: received=1 broadcast_sent=2 requested=2
Bus: BusRd=4 BusRdX=2 BusUpgr=0 WB=1
Traffic (data only): read=192 (0.19 KB) written=32 (0.03 KB) total=0.000214 MB
MESI transitions:
  I->E=2 I->S=2 I->M=2
  E->M=0 E->S=0 E->I=0
  S->M=0 S->I=0
  M->S=1 M->I=1

[PE3 statistics]
Reads: hits=14 misses=7 total=21
Writes: hits=0 misses=2 total=2
Totals: hits=14 (60.87%) misses=9 (39.13%) accesses=23
Invalidations: received=2 broadcast_sent=0 requested=2
Bus: BusRd=7 BusRdX=2 BusUpgr=0 WB=1
Traffic (data only): read=288 (0.28 KB) written=32 (0.03 KB) total=0.000305 MB
MESI transitions:
  I->E=5 I->S=2 I->M=2
  E->M=0 E->S=1 E->I=0
  S->M=0 S->I=0
  M->S=0 M->I=2
```

Fig. 6. Individual PE statistics of the execution of a dot product program with a size 16 vector.


daniel-cob@daniel-cob-A320M-H: ~/Documents/arquitectura de computa...

```
[Statistics summary]
Hit rates per PE:
  PE0: accesses=14 hits=8 misses=6 hit=57.14% miss=42.86%
  PE1: accesses=14 hits=8 misses=6 hit=57.14% miss=42.86%
  PE2: accesses=14 hits=8 misses=6 hit=57.14% miss=42.86%
  PE3: accesses=23 hits=14 misses=9 hit=60.87% miss=39.13%
Total: accesses=65 hits=38 misses=27 hit=58.46% miss=41.54%
Invalidations per PE:
  PE0: received=2 broadcast_sent=0 requested=2 total=2
  PE1: received=2 broadcast_sent=1 requested=2 total=3
  PE2: received=1 broadcast_sent=2 requested=2 total=3
  PE3: received=2 broadcast_sent=0 requested=2 total=2
Total: received=7 broadcast_sent=3 total=10
Bus traffic per PE (data only) (KB):
  PE0: BusRd=4 BusRdX=2 BusUpgr=0 WB=2 Traffic=0.25
  PE1: BusRd=4 BusRdX=2 BusUpgr=0 WB=1 Traffic=0.22
  PE2: BusRd=4 BusRdX=2 BusUpgr=0 WB=1 Traffic=0.22
  PE3: BusRd=7 BusRdX=2 BusUpgr=0 WB=1 Traffic=0.31
Total bus traffic (data only): 1.00 KB (0.000977 MB)

[Memory statistics]
Accesses: reads=18 writes=8 total=26
Traffic: read=576 (0.56 KB) written=256 (0.25 KB) total=0.000793 MB
Accesses per PE:
  PE0: reads=4 writes=2 total=6
  PE1: reads=5 writes=1 total=6
  PE2: reads=2 writes=3 total=5
  PE3: reads=7 writes=2 total=9

[Bus statistics]
Transactions: BUS_RD=19 BUS_RDX=8 BUS_UPGR=0 BUS_WB=5 Total=32
Coherence: broadcast_invalidations=3
Traffic: data=1024 (1.00 KB) control=408 (0.40 KB) total=1432 (1.40 KB, 0.001366
 MB)
  Control breakdown: base=384 (0.38 KB) invalidations=24 (0.02 KB)
Usage per PE (transactions and %):
  PE0: 8 (25.00%)
  PE1: 7 (21.88%)
  PE2: 7 (21.88%)
  PE3: 10 (31.25%)
Efficiency: bytes/transaction=44.75 reads=59.38% writes=40.62%
```

Fig. 7. Global system statistics during the execution of the dot product.

Overall, the results confirm that the multiprocessor model behaves consistently with MESI coherence theory. The caches achieved positive hit ratios, the bus effectively serialized requests, and the memory module maintained consistency without unnecessary accesses. Misalignment experiments highlighted the importance of data locality for performance. Together, these findings validate both the correctness and the efficiency of the simulated architecture.

IV. Conclusions

The development and simulation of a MESI-coherent multiprocessor model successfully demonstrated both the functional and performance characteristics expected from shared-memory architectures. The four-PE system with private caches, connected through a Round-Robin–scheduled bus, achieved correct synchronization and data consistency during the execution of a parallel dot product program.

The results showed that the implemented cache architecture effectively reduced the number of memory accesses by exploiting spatial and temporal locality. The MESI protocol maintained coherence efficiently, as evidenced by the low percentage of invalidation traffic and the high cache hit ratio obtained across all PEs. The master PE exhibited a slightly higher workload due to the final reduction phase, a predictable behavior consistent with parallel reduction operations.

The analysis of misaligned vector data revealed that even small deviations from alignment significantly impact system performance. Increased bus activity and coherence traffic highlighted the sensitivity of cache-based systems to data organization, reinforcing the importance of memory alignment and locality in high-performance parallel computation.

Overall, the proposed model achieved its goals by providing a functional and analyzable platform to study cache coherence mechanisms in a multiprocessor environment. The results validate the theoretical foundations of the MESI protocol and illustrate their practical implications in performance-critical scenarios.

V. Recommendations

- In future implementations of a multiprocessor model the usage of appropriate types of data in C should be corrected to make traffic calculations more realistic.
- A different type of application, other than dot product, should be implemented to observe more types of MESI transitions and observe its performance.
- Consider scaling the number of PEs to observe the limits in parallelism of the programs.

References

[1] John L Hennessy y David A Patterson. Computer Architecture: A Quantitative Approach. Elsevier, 2017