Shared memory project 2025

By: Aya Fodi, Dima Omar

Prof. Yosi Ben Asher

University of Haifa

Introduction

In this project, a shared memory architecture for manycore systems was developed and implemented. The design is based on an existing theoretical model in which the address space is divided among dual-port memory modules connected via a Butterfly communication network.

The project aims to address common challenges in such systems, including scalability issues, memory access bottlenecks, and the high cost of maintaining memory consistency (cache coherency), which are typical in classical shared memory architectures.

To address these issues, the following solutions were implemented:

- Use of a Butterfly network as a combinational circuit to route read/write requests in a parallel and efficient manner.
- Integration of an internal buffer mechanism within each switch to manage collisions, including dynamic priority handling.
- A hashing mechanism for memory addresses to distribute requests across memory modules, aiming to reduce collisions and alleviate congestion.

The system was evaluated through comprehensive simulation of various scenarios: write-only operations, read-only operations, requests to a single fixed address versus distributed requests — while analyzing parameters such as:

- Number of collisions in the network and memory modules
- Number of dropped packets
- Average and maximum latency
- Impact of load size (T) on system performance

The results demonstrated that the implemented architecture is stable, scalable, and robust even under extreme conditions. Performance improved with a larger number of cores and when memory requests were well-distributed. The use of hashing and an intelligent collision management mechanism enabled the system to maintain high throughput and low latency even under heavy and distributed workloads.

System Architecture

The system is designed from $N = 2^k$ cores, where each core sends T read or write requests to a shared memory. Each request goes through the following stages:

1. **Address Hashing:** Every request is passed through a hashing function that maps the address to a specific memory module — to balance the load and prevent local contention.
2. **Forward Butterfly Network:** The requests enter a Butterfly network composed of $\log(N) - 1$ layers, each containing $\frac{N}{2}$ switches of type switch2x2.

Each switch receives two inputs and produces two outputs, routing packets based on their target address and priority.

3. **Memory Access:** At the output of the forward network, there are $\frac{N}{2}$ dual-port RAM modules that serve the concurrent memory requests.

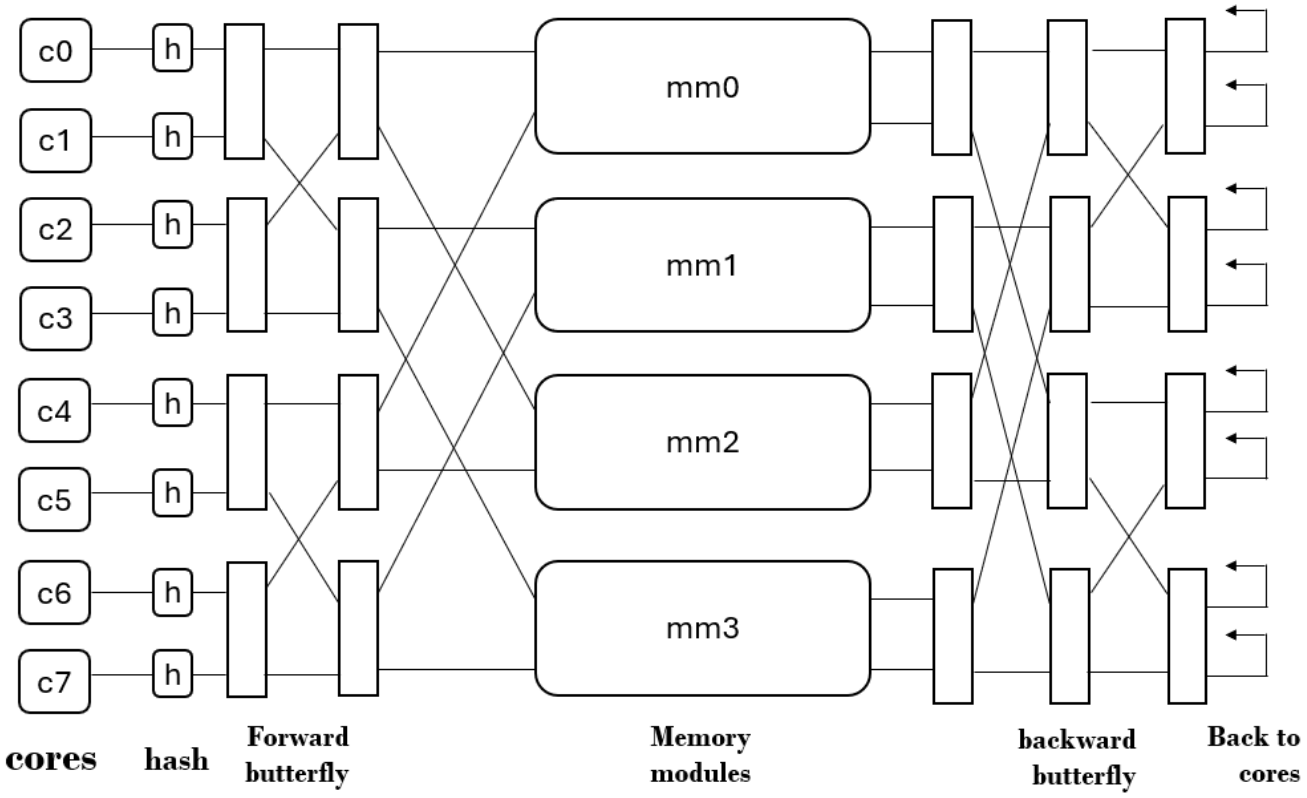
4. **Back-Packet Generation:** For every request, a corresponding back-packet is generated. This packet includes the read result, the core ID, and a success/failure indicator.

5. **Backward Butterfly Network:** The response packets are sent back to the cores through a backward Butterfly network consisting of $\log(N)$ layers, each again with $\frac{N}{2}$ switches.

6. The backward network is responsible for delivering the memory response back to the originating core.

7. **Packet Propagation:** Within both the forward and backward networks, packets are routed combinationally and without clock dependence. Only the entry/exit points of buffers and memory modules are synchronized with the system clock.

Example of Network Structure for $k = 3$:



Packets Structure:

Forward Packet:[RW | Module_ID | Local_Address | Priority | DATA |CORE_ID]

- **RW (1 bit):** Indicates whether the operation is a read (0) or a write (1).
- **Module_ID:** Identifies the target memory module. Width: MOD_ID_BITS = K_LOG2 - 1.
- **Local_Address:** The internal address within the memory module.
- **Priority:** A 2-bit field representing the request's priority.
- **DATA:** The actual data being read or written.
- **CORE_ID:** Identifies the core that issued the request.

Backward Packet: [CORE_ID | Priority | Success | DATA]

- **CORE_ID:** The core to which the packet is returned.
- **Priority:** The priority of the original request.
- **Success:** A 1-bit flag indicating whether the request was successful.
- **DATA:** The result retrieved from memory (in the case of a read operation).

Hash Function Used:

We implemented a function named xor_shift_hash, which takes a full memory address and applies XOR operations with shifted versions of the address (by 5 and 11 bits) to mix the bits and achieve better randomness.
After the mixing, only the least significant bits (LSBs) are extracted to determine the target memory module ID.
The goal is to achieve a uniform distribution of requests across the memory modules, thereby reducing collisions and bottlenecks — while using a simple and hardware-efficient function.

Request Generation and Transmission Process:

For each core, a predefined list of T read/write requests to various memory addresses is generated.
The simulation operates in discrete clock cycles, where new requests are transmitted according to the following conditions:

- **Success-Based Transmission:** A core sends its next request only if it previously received a response packet with the success bit set$(\mathrm{Success} = 1)$, if its previous request was dropped due to congestion its sends it request again.
- **Full Parallel Transmission:** In every clock cycle, a vector of all active requests from all cores is transmitted. This vector has a total width of $\mathrm{N} * \mathrm{PACKET\_W}$ and contains all packets being sent during that cycle.
- **Valid Bus:** Alongside the packet vector, a Valid vector of width N is sent, indicating whether each core's packet is valid in the current cycle (valid[i] = 1) or not (valid[i] = 0).
- **Core Stalling Condition:** A core that did not receive a valid response or is in a waiting state will not send a new packet until it is allowed to do so based on the above rules.

Construction of the Forward Butterfly Network:

The forward network is constructed using a classic Butterfly topology, allowing each individual packet to follow a predetermined path to its destination — based on the Module_ID field within the packet.

Network Structure:

- The network consists of $\log(N) - 1$ layers, each containing $\frac{N}{2}$ switch2x2 units.
- Each switch receives two inputs and provides two outputs, deciding the routing path based on a specific bit of the destination module ID.

Routing and Bit-Revealing Logic:

- In layer *i* of the network, the output decision is determined by the *i-th* bit of the Module_ID field in the packet.
- That is, each packet "reveals" one bit of its target destination at each stage of the network, corresponding to the depth of its progress.
- Between each pair of layers, a fixed permutation of the wires is performed — based on the classic Butterfly interconnection pattern.

Construction of the Backward Butterfly Network:

The backward network is constructed similarly to the forward Butterfly network. However, the key differences are:

- The routing decisions are based on the Core_ID field instead of the Module_ID.
- The network consists of $\log(N)$ layers (one more than the forward network).

Network Output:

- In each clock cycle, the results are output in the out_flat structure, with a total width of N * BACK_PACKET_W, containing all the returning packets.
- The valid_back_out field indicates which cores received a valid response.
- The dropped_core_bus field reports which cores experienced a packet drop.
- The total_collisions field counts the total number of collisions detected across all switches.

Module: switch2x2:

Inputs:

- a, b: Two packets of width wir_W (can be either PACKET_W or BACK_PACKET_W)
- va, vb: Valid bits for each packet
- sel_a, sel_b: Selection bits indicating the target output for each packet
- clk: Clock signal

Outputs:

- y0, y1: Packet outputs
- v0, v1: Valid bits for the outputs
- dropped_core_vector: Bit vector indicating which cores experienced a drop
- collision_detected: A flag indicating that a collision occurred

Internal Buffer:
The buffer stores a single packet with the following associated fields:

- buffer: The packet content
- buffer_sel: The designated output port for the buffered packet
- buffer_pri: The updated priority value of the buffered packet

Collision Management:

The switch2x2 module is responsible for routing two incoming packets to two output ports, while handling potential collisions using a priority-based mechanism and an internal buffer.

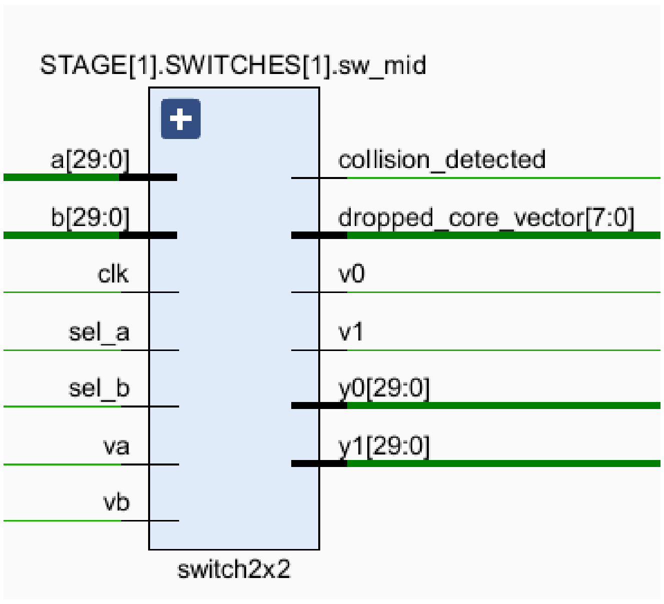
Collision Handling Policy:

1. **Buffer Has Priority:**
   When a packet already exists in the buffer (buffer_valid = 1), it is always given priority and routed to its designated output (y0 or y1) before any new incoming packets are considered.
2. **Priority Between New Inputs:**
   If both incoming packets (a and b) target the same output port:
   - Their priorities are compared.
   - The packet with the higher priority is routed to the output.
   - If the priorities are equal, packet a (which arrived first) is given priority.
3. **Buffering and Priority Boost:**
   The packet that was not selected is stored in the internal buffer of the switch, and its priority is incremented.
   This ensures that packets blocked multiple times will eventually gain priority in future cycles.
4. **Triple Collision (3 Packets):**
   When a buffered packet exists and both incoming packets collide with it:
   - The buffered packet is forwarded.
   - One of the two new packets is stored in the buffer (chosen based on the policy above).
   - The third packet is dropped and flagged in the dropped_core_vector.
5. **Collision Detection:**
   The collision_detected signal is asserted when:
   - Both new packets target the same output, **or**
   - One or both of the new packets conflict with the output destination of the buffered packet.

Collision Aggregation in the Butterfly Network:

Within the Butterfly network, all dropped_core_vector signals generated by the switches are combined using a bitwise OR operation. The resulting vector indicates which cores experienced packet drops and is output from the network as a single aggregated signal.

In parallel, each switch maintains a collision_detected flag. For every clock cycle, a counter accumulates the total number of switches that raised this flag. The resulting count is transmitted through a multi-bit signal named total_collisions, which reflects the total number of collisions detected across the entire network during that cycle.



Module: dual_port_ram

This module represents an internal dual-port memory block, allowing two independent requests—one from **Port A** and one from **Port B**—to access the memory concurrently in the same clock cycle. The module detects conflicts between the requests, resolves their priority, and constructs a **back-packet** response that is returned to the requesting core, including read/write data and a success status.

Port A Signals:

- we_a: Write enable signal
- local_addr_a: Internal address within the memory module
- pri_a: Request priority
- wdata_a: Data to be written
- core_id_a: Core ID of the requester
- valid_a: Indicates whether the request is valid
- valid_a_out: Output valid signal after processing
- rpkt_a: Back-packet response

Port B Signals:
Fully parallel to Port A, with equivalent signals: we_b , local_addr_b, pri_b, wdata_b, core_id_b, valid_b, valid_b_out, and rpkt_b.

Shared Inputs: clk: Clock signal

Conflict Flags:

- same_addr: Indicates both ports target the same address
- ww_conflict: Write–write conflict on the same address
- rw_hazard_ab / rw_hazard_ba: Read/write hazards between Ports A and B

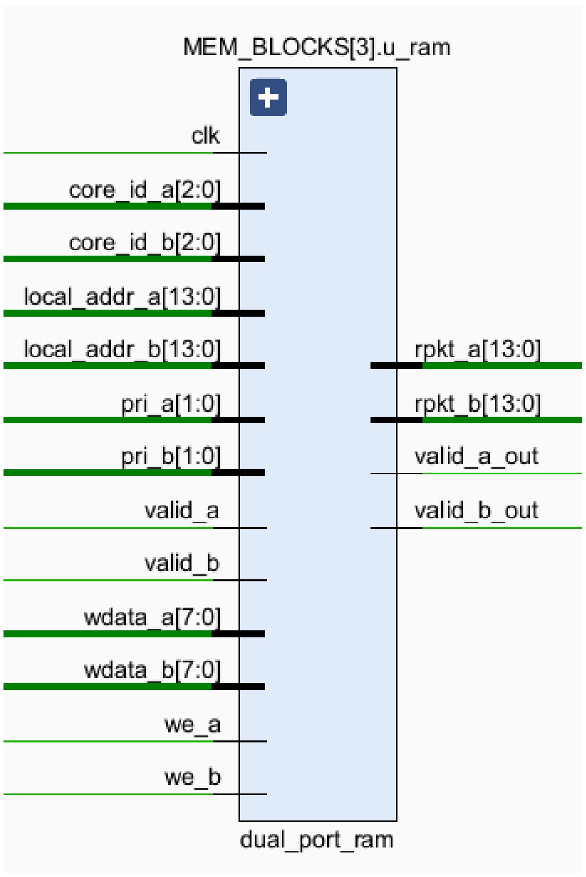
Back-Packet:

The back-packet returned to the core contains not only the data retrieved from memory (in case of a read) but also a success/failure flag. If the request fails (due to conflict), the core will attempt to resend it in the next cycle.

Operational Logic:

- **Write & Read:**
  If a port issues a valid request (valid = 1) and the write-enable signal (we) is asserted, the provided data is written to the memory.
  Simultaneously, the memory is always read from the selected address, and the result is stored in a register (dout).
- **Response Generation:**
  If the request is valid, the valid_out signal is asserted, and a back-packet is generated containing the data read from memory along with a success status.
- **Conflicts Between Ports A and B:**
  If both requests target the same address in the same cycle:

  - **WW Conflict (Double Write):**
    When both ports attempt to write to the same address, **Port A** is given priority, and the write request from Port B is rejected.
  - **Read/Write Hazard:**
    If one port performs a write and the other a read to the same address in the same cycle, the value returned by the read operation is the **old value** from memory, **prior to the write**.
    This choice ensures consistency and avoids undefined behavior in simulation.
    The situation is flagged using the rw_hazard_ab and rw_hazard_ba signals, allowing the system to detect and handle such hazards appropriately.


MEM_BLOCKS[3].u_ram
clk
core_id_a[2:0]
core_id_b[2:0]
local_addr_a[13:0]
local_addr_b[13:0]
pri_a[1:0]
pri_b[1:0]
valid_a
valid_b
wdata_a[7:0]
wdata_b[7:0]
we_a
we_b
rpkt_a[13:0]
rpkt_b[13:0]
valid_a_out
valid_b_out
dual_port_ram

Tests and Results:

The number of requests was set to 20 per core:

# Cores	2^3	2^4	2^5	2^6	2^7	2^8	2^9	2^{10}
Period (ns)	6.6	8.7	10.1	12.7	13.2	14.6	16	19
Frequency (MHz)	151.5	115	99	78.74	76	68.5	62.5	
Switch collisions	97	269	629	1482	3541	8162	17666	37021
Total drops	6	7	28	61	160	393	763	1708
Average Latency	2.68	2.89	3.07	3.27	3.52	3.76	3.88	3.98
Max Latency	8	9	8	13	15	21	21	28
Logic Power (W)	0.013	0.026	0.056	0.111	0.184	0.368		
Clocks Power (W)	_0.009	0.018	0.041	0.116	0.369	1.411		
Signals Power (W)	0.015	0.028	0.049	0.097	0.165	0.350		
LUTs	2433	7446	25038	64683	162458	467587	1903113	
Registers	608	2239	8384	30094	125312	546304	2355712	

Synthesis and Runtime Analysis – Impact of Core Count:

- **Good Network Scalability:**
  As the number of cores increases (from 8 up to 1024), the average latency rises only moderately — from 2.68 to 3.98 clock cycles — indicating stable scalability even under heavy traffic.
- **Efficient Collision Handling:**
  Although the number of collisions in switches increases significantly (by a factor of ~380), the number of dropped packets remains relatively low compared to the load, demonstrating the network's ability to reroute traffic and maintain continuous flow.
- **Controlled Maximum Latency:**
  The maximum observed latency remains within acceptable limits (28 cycles at 1024 cores), showing that even in edge cases, there is no performance collapse or severe network congestion.
- **Maintained Timing Viability:**
  Despite a reduction in operating frequency (from 151 MHz to 62.5 MHz), the network maintains stable timing and enables correct functionality even at large scale — without requiring deep pipelining or partitioning into sub-networks.
- **Linear Resource Utilization:**
  The use of LUTs and registers increases in a predictable and linear manner with network size, consistent with the Butterfly structure of $\log(N)$ stages × N cores — with no abnormal spikes or resource waste.
- **Effective Buffer Usage and Minimal Delay:**
  Thanks to smart buffer management in the switches and dynamic priority boosting of delayed packets, the network avoids repeated drops and enables smooth congestion resolution.
- **Architectural Scalability Proven:**
  The results demonstrate that the system handles core count scaling effectively, maintaining a balance between resource usage, performance, and response time.
  Moreover, the ratio between maximum latency and the number of cores **decreases** as the system scales, indicating **improved relative performance** under load.
  This suggests that, while latency grows in absolute terms, the per-core overhead becomes smaller — reflecting better efficiency and throughput as the system scales up.

Test Environment: Distributed read requests using a hash function.

#cores = 2^8

T (# of request)	1	3	10	20	50	120
Switch collisions	256	1392	4606	8162	21400	60838
Total drops	64	190	266	393	1160	5207
Average Latency	3.5	4.307	4.015	3.764	3.880	4.331
Max Latency	5	15	16	21	28	62

Results:

- **Scalability Under Load:**
  Even under high load (T=120), the average latency remains low and stable — between 3.5 and 4.3 clock cycles.
- **Efficient Collision Handling:**
  Although switch collisions reach up to ~60,000, the number of dropped packets remains low (≤ 5,207), indicating smart conflict resolution.
  Less than 10% of collisions result in actual drops.
- **Controlled Max Latency:**
  While maximum latency increases under load (up to 62 cycles), most requests complete in a short time.
- **Bottleneck Located in the Network, Not Memory:**
  Optimization of the switching logic could further improve performance.
- **Improved Efficiency with Load:**
  The ratio MaxLatency / T decreases with larger T values, indicating better per-request latency under high traffic.
- **Conclusion:**
  The network demonstrates strong stability, scalability, and responsiveness — even with hundreds of cores and high traffic levels.

Test Environment: Distributed write requests generated using a hash function for address mapping.

#cores = 2^8

T (# of request)	1	3	10	20	50	120
Switch collisions	256	1392	4606	8162	21400	60838
Total drops	64	190	266	393	1160	5207
Average Latency	3.5	4.307	4.015	3.764	3.880	4.331
Max Latency	5	15	16	21	28	62
memory collisions	0	0	0	0	0	0

Performance Analysis – Memory Behavior

- **Zero Memory Collisions:**
  Across all tested loads (T=1 to T=120), **no memory module collisions** were observed.
- **Explanation:**
  When memory addresses are well-distributed (via hashing), the probability of two writes targeting the same module in the same cycle is extremely low — even with 256 cores writing simultaneously.
- **Network as the Main Bottleneck:**
  All observed collisions occurred in the **switching network**, not in the memory itself. The memory modules remained congestion-free.
- **Critical Advantage of Hashing:**
  These results highlight the importance of address hashing and uniform distribution across memory modules — which protects the system from memory "hot spots".
- **Practical Implication:**
  Even under full parallel writes by all cores, **no single memory module is overloaded**.
  (Although a hardware mechanism exists to resolve write conflicts by prioritizing **Port A** over **Port B** in case of address conflict, this scenario did not occur here. It will be tested under more extreme conditions in the next experiment.)

Test Environment: Read requests targeting the **same memory address** for all cores.

#cores = 2^8

T (# of request)	1	3	10	20	50	120
Switch collisions	1165	3703	12363	25284	62774	149451
Total drops	296	1048	3623	7612	19417	46188
Average Latency	8.921	9.585	9.698	9.953	9.976	9.910
Max Latency	18	42	131	198	451	425

Performance Analysis:

- **Severe Network Congestion:**
  Switch collisions increased dramatically — up to **149,451** (compared to 60,838 under hashed distribution).
- **High Drop Rate:**
  A total of **46,188 drops** were recorded (vs. 5,207 in the hashed case) — nearly **9× more**.
- **Poor Latency Behavior:**
  Average latency almost **doubled** (9.91 vs. 4.33), while **maximum latency** reached up to **425 clock cycles**.
  However, the average remains relatively stable across different values of T.
- **Resilient Network Design:**
  Despite the extreme load, the system **does not crash** — thanks to the use of internal buffers and retransmission mechanisms, the network continues functioning under stress.

Test Environment: write requests targeting the **same memory address** for all cores.

#cores = 2^8

T (# of request)	1	3	10	20	50	120
Switch collisions	15375	46413	155035	310391	776596	1866089
Total drops	8643	26367	88230	176527	441399	1058766
Average Latency	129.742	131.180	131.548	131.644	131.707	131.733
Max Latency	257	755	2501	4987	12495	29842
memory collisions	87	320	1131	2227	5532	13567

Performance Analysis – Extreme Load Scenario:

- **Network Bottleneck:**
  Over **1.8 million** switch collisions were recorded for T=120.
- **Memory Bottleneck:**
  More than **13,000 memory collisions**, due to all requests targeting the same memory module.
- **Very High Latency:**
  Average latency reached **~131 clock cycles**, with **maximum latency nearing 30,000**.
- **Massive Packet Drops:**
  Over **1 million drops**, caused by repeated retransmissions that further increased network load.
- **Despite It All – The Network Survives:**
  Internal mechanisms (buffering, priority escalation, retries) **prevent system failure**, but performance remains significantly degraded under this extreme condition.