SCIENCES

Electronics Engineering, Field Director – Francis Balestra

Design Methodologies and Architecture, Subject Head – Ahmed Jerraya

Multi-Processor System-on-Chip 1

Architectures

Coordinated by
Liliana Andrade
Frédéric Rousseau

Color Section

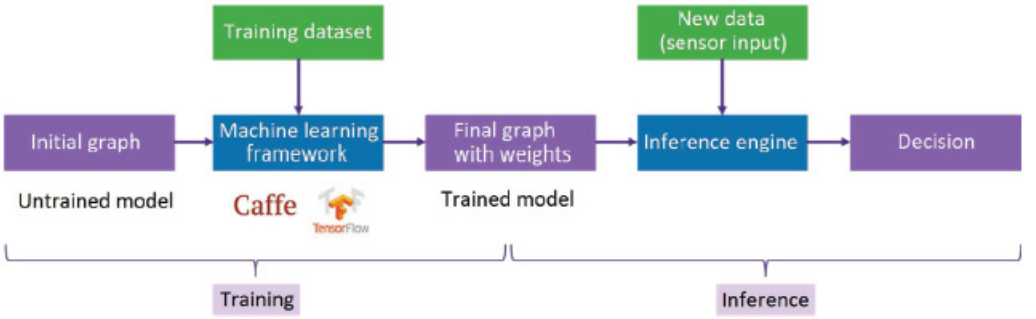


Figure 1.1. *Training and inference in machine learning*

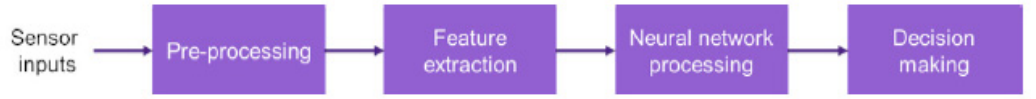


Figure 1.2. *Different types of processing in machine learning inference applications*

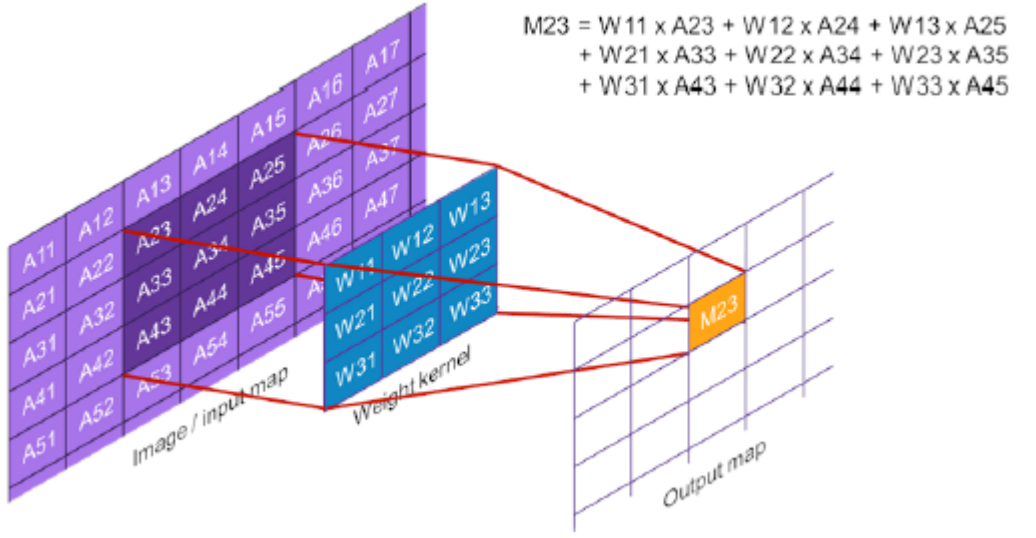


Figure 1.3. *2D convolution applying a weight kernel to input data to calculate a value in the output map*

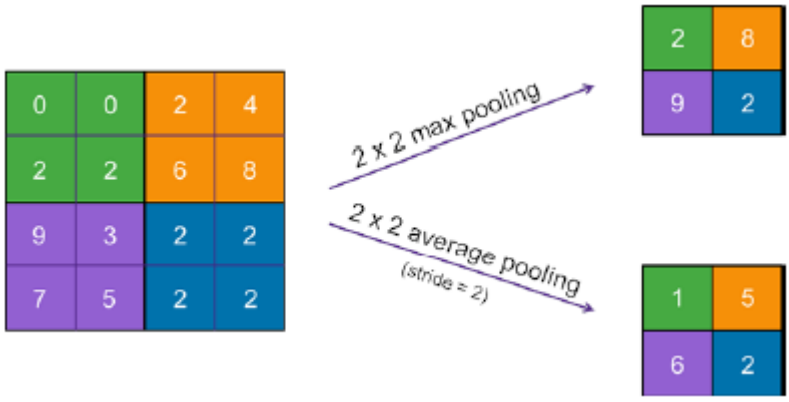


Figure 1.4. *Example pooling operations: max pooling and average pooling*

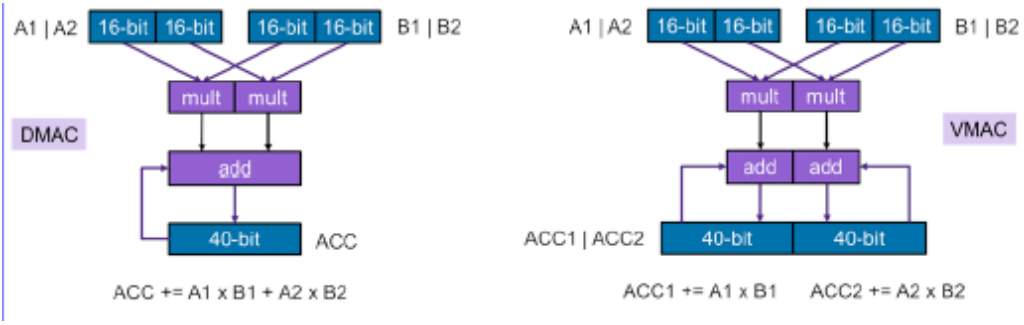


Figure 1.5. *Two types of vector MAC instructions of the ARC EM9D processor*

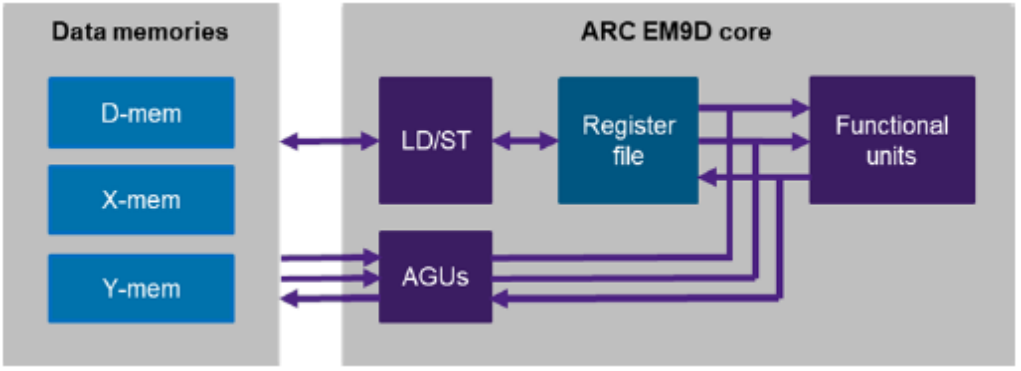


Figure 1.6. *ARC EM9D processor with XY memory and address generation units*

```
    ...       ; AGU 0 is used for input data
              ; AGU 1 is used for weights
    SR  ...   ; setup AGU 0 for loading next 2x16-bit vector
    SR  ...   ; setup AGU 1 for loading next 2x8-bit vector
    ...
    LP        lpend
    DMACHBL 0, %agu_u0, %agu_u1 ;
lpend: ...
```

Figure 1.7. *Assembly code generated from MLI C-code for a fully connected layer with 16-bit input data and 8-bit weights*

```
    ...       ; AGU 0 and AGU 1 are used for input data
              ; (one data pointer with two modifiers)
              ; AGU 2 is used for weights
    SR  ...   ; setup AGU 0 for loading next two 16-bit values
    SR  ...   ; setup AGU 1 for loading two 16-bit input values
              ; at next row of input data
    SR  ...   ; setup AGU 2 for loading next 8-bit value with
              ; sign extension & replication
    ...
    LP          lpend
    VMAC2HNFR   0, %agu_u0, %agu_u2 ;
    VMAC2HNFR   0, %agu_u0, %agu_u2 ;
    VMAC2HNFR   0, %agu_u1, %agu_u2 ;
lpend: ...
```

Figure 1.8. *Assembly code generated from MLI C-code for 2D convolution of 16-bit input data and 8-bit weights*

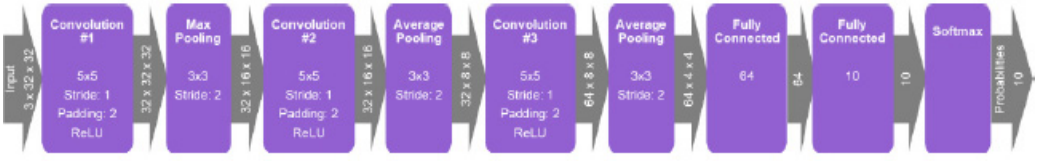


Figure 1.9. *CNN graph of the CIFAR-10 example application*

```
mli_krn_permute_fx8(&input, &permute_hwc2chw_cfg, &ir_Y);

ir_X.el_params.fx.frac_bits = CONV1_OUT_FRAQ;
mli_krn_conv2d_chw_fx8_k5x5_str1_krnpad(&ir_Y, &L1_conv_wt,
  &L1_conv_b, &conv_cfg, &ir_X);
mli_krn_maxpool_chw_fx8_k3x3(&ir_X, &pool_cfg, &ir_Y);

ir_X.el_params.fx.frac_bits = CONV2_OUT_FRAQ;
mli_krn_conv2d_chw_fx8_k5x5_str1_krnpad(&ir_Y, &L2_conv_wt,
  &L2_conv_b, &conv_cfg, &ir_X);
mli_krn_avepool_chw_fx8_k3x3_krnpad(&ir_X, &pool_cfg, &ir_Y);

ir_X.el_params.fx.frac_bits = CONV3_OUT_FRAQ;
mli_krn_conv2d_chw_fx8_k5x5_str1_krnpad(&ir_Y, &L3_conv_wt,
  &L3_conv_b, &conv_cfg, &ir_X);
mli_krn_avepool_chw_fx8_k3x3_krnpad(&ir_X, &pool_cfg, &ir_Y);

ir_X.el_params.fx.frac_bits = FC4_OUT_FRAQ;
mli_krn_fully_connected_fx8(&ir_Y, &L4_fc_wt, &L4_fc_b, &ir_X);

ir_Y.el_params.fx.frac_bits = FC5_OUT_FRAQ;
mli_krn_fully_connected_fx8(&ir_X, &L5_fc_wt, &L5_fc_b, &ir_Y);
mli_krn_softmax_fx8(&ir_Y, &output);
```

Figure 1.10. *MLI code of the CIFAR-10 inference application*

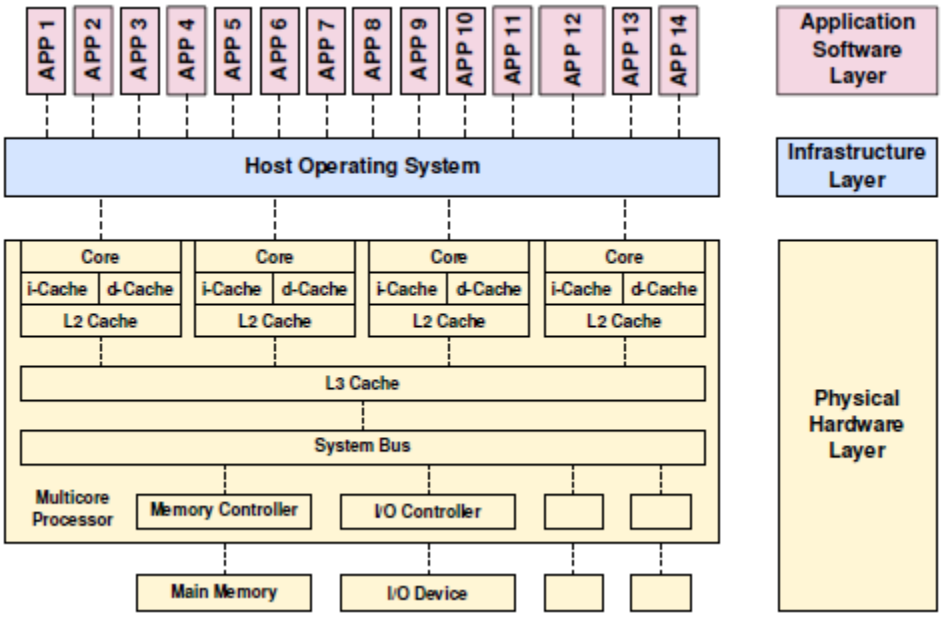


Figure 2.1. *Homogeneous multi-core processor (Firesmith 2017)*

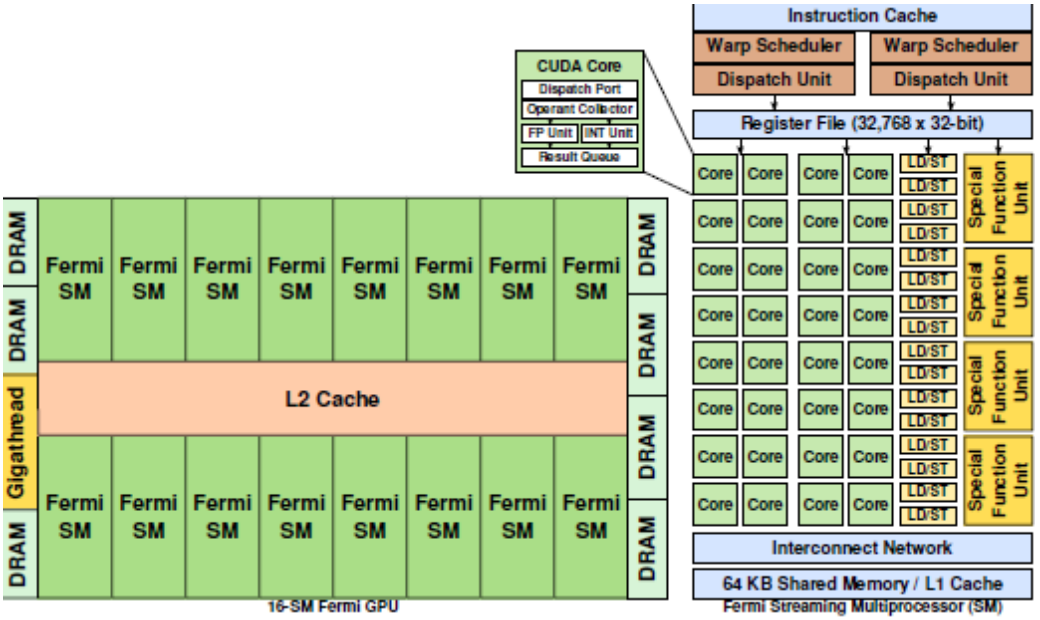


Figure 2.2. *NVIDIA fermi GPGPU architecture (Huang et al. 2013)*

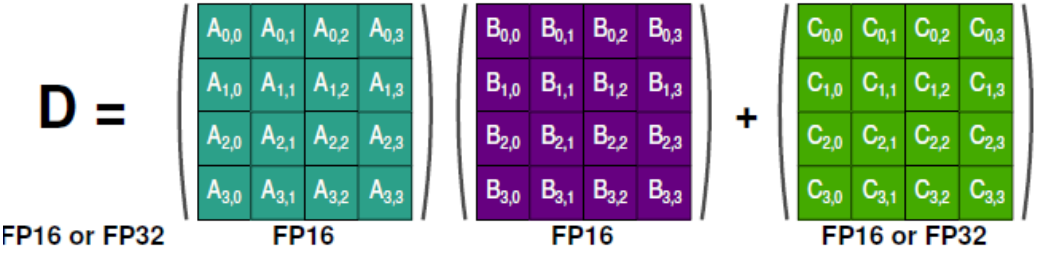


Figure 2.3. *Operation of a Volta tensor core (NVIDIA 2020)*

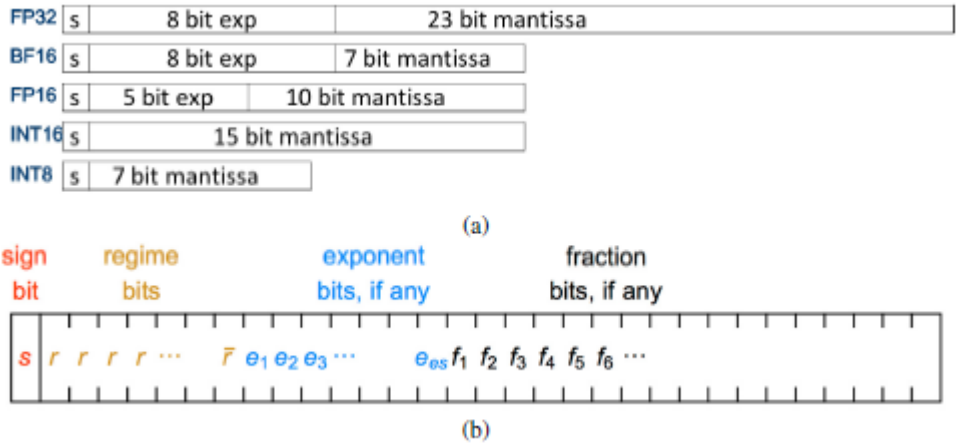


Figure 2.4. *Numerical formats used in deep learning inference (adapted from Gustafson (2017) and Rodriguez et al. (2018))*

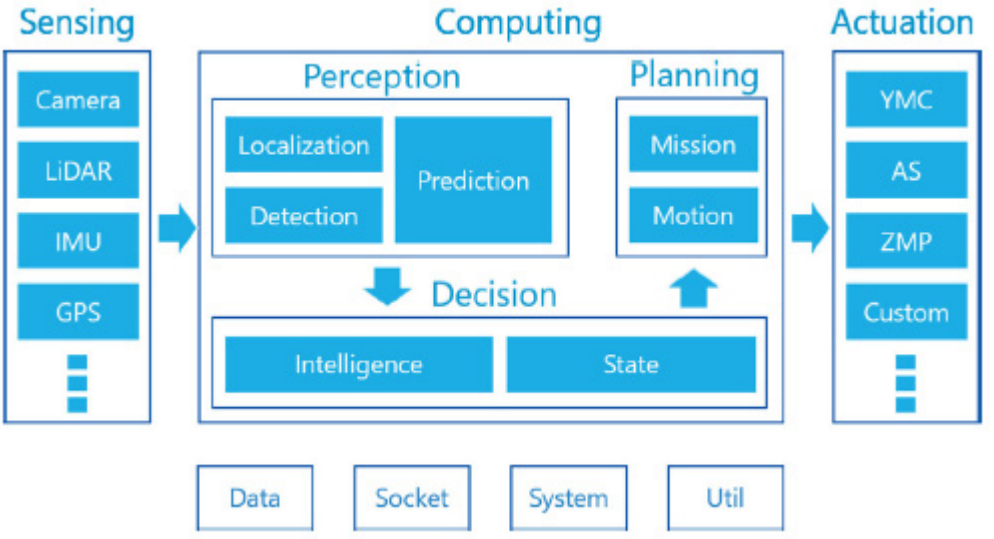


Figure 2.5. *Autoware automated driving system functions (CNX 2019)*

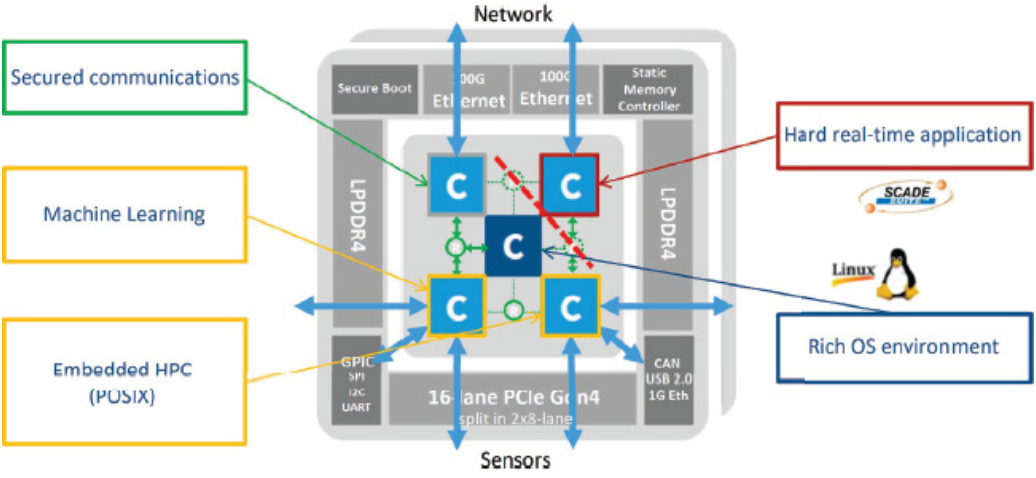


Figure 2.6. *Application domains and partitions on the MPPA3 processor*

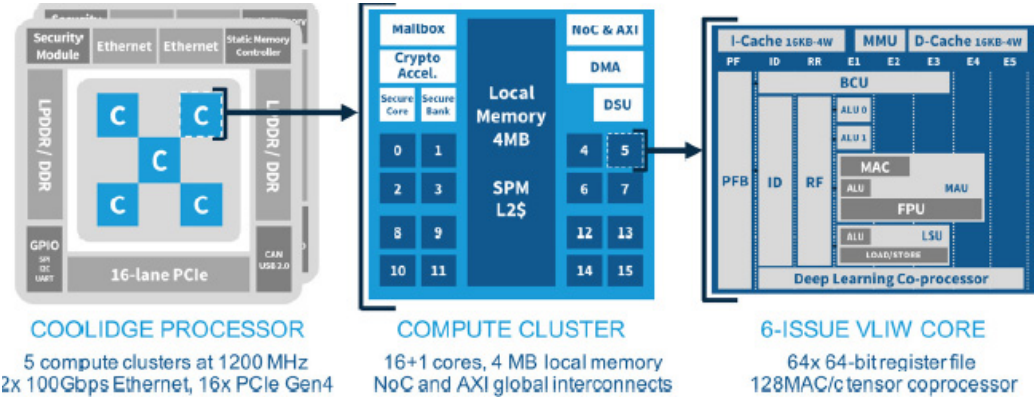


Figure 2.7. *Overview of the MPPA3 processor*

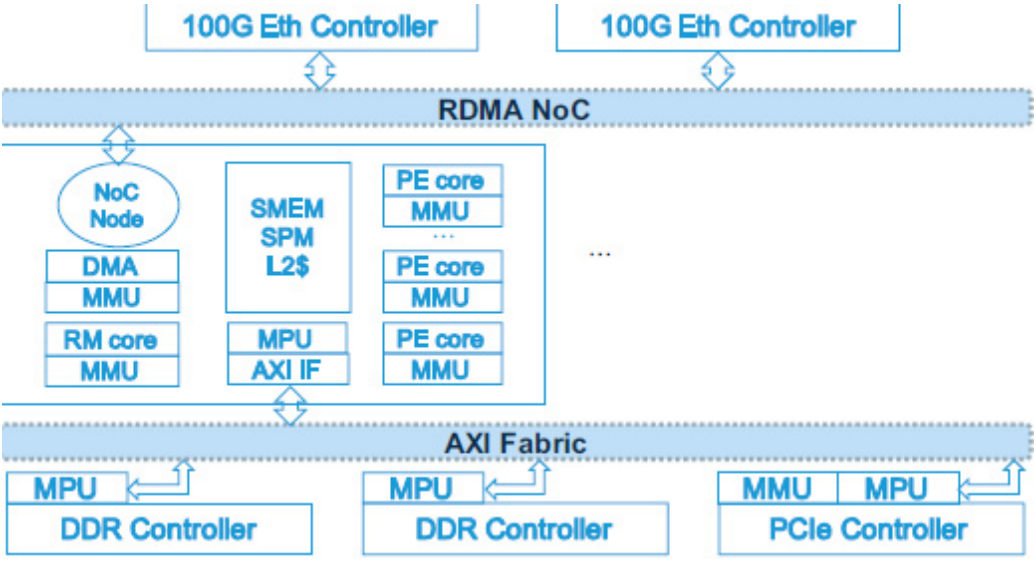


Figure 2.8. *Global interconnects of the MPPA3 processor*

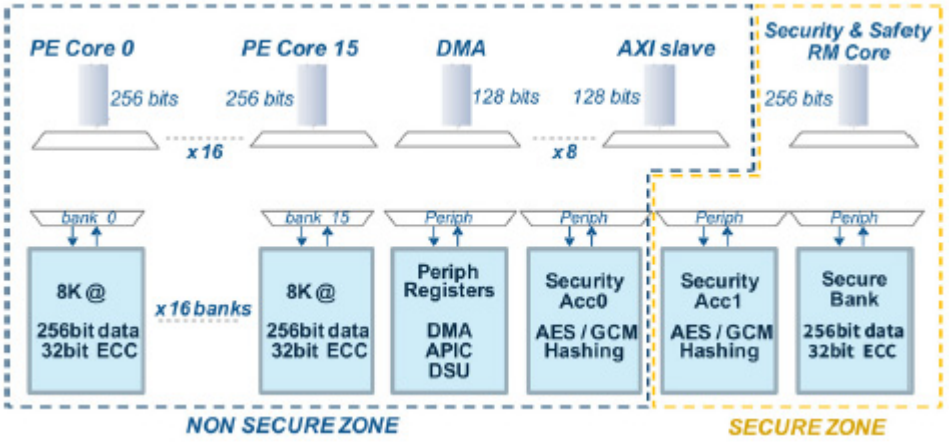


Figure 2.9. *Local interconnects of the MPPA3 processor*

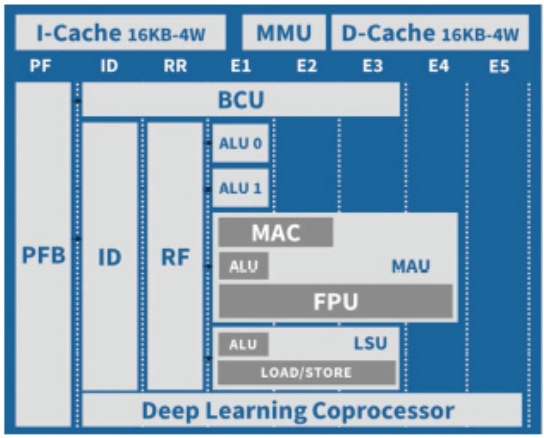


Figure 2.10. *VLIW core instruction pipeline*

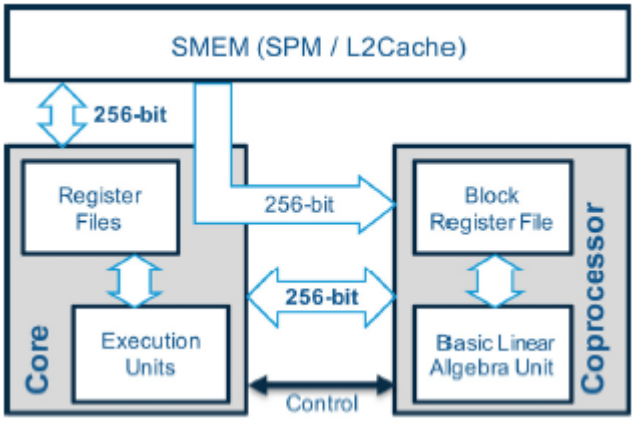


Figure 2.11. *Tensor coprocessor data path*

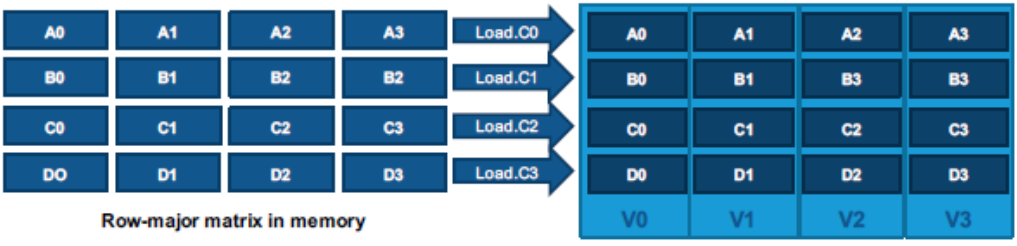


Figure 2.12. *Load-scatter to a quadruple register operand*

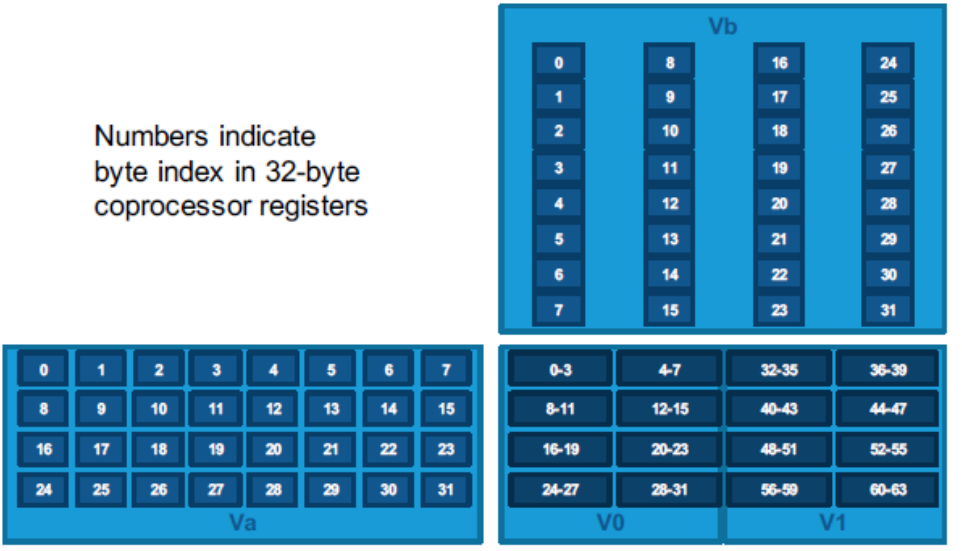


Figure 2.13. *INT8.32 matrix multiply-accumulate operation*

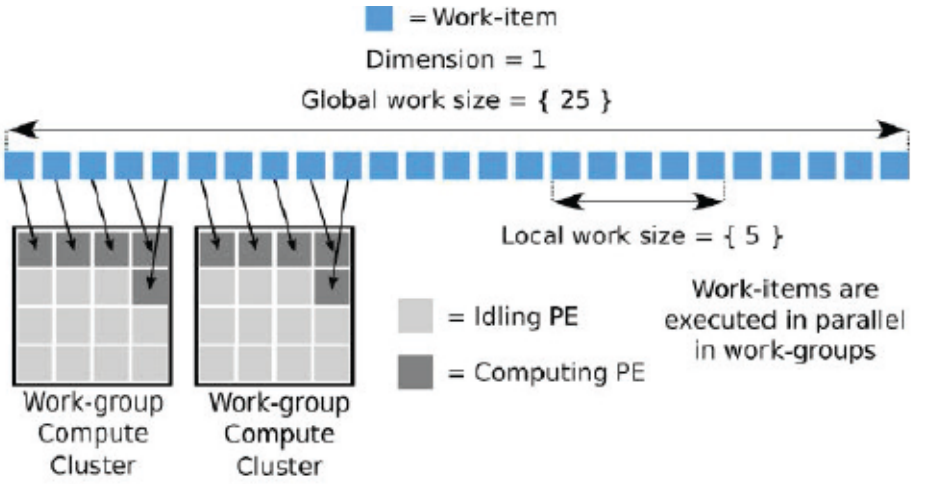


Figure 2.14. *OpenCL NDRange execution using the SPMD mode*

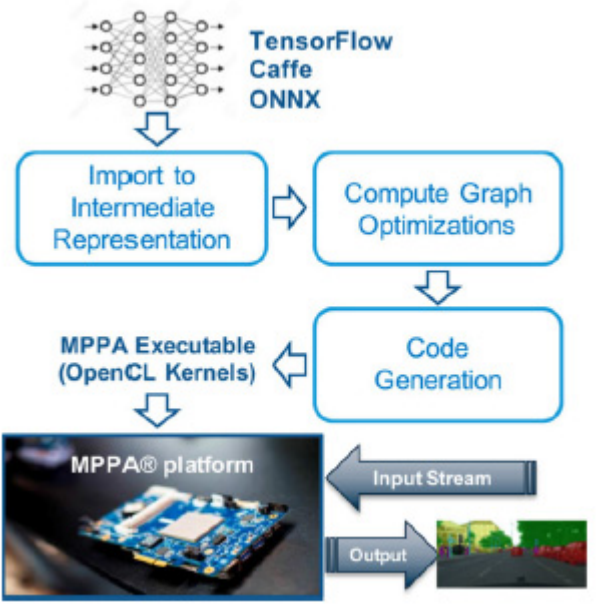


Figure 2.15. *KaNN inference code generator workflow*

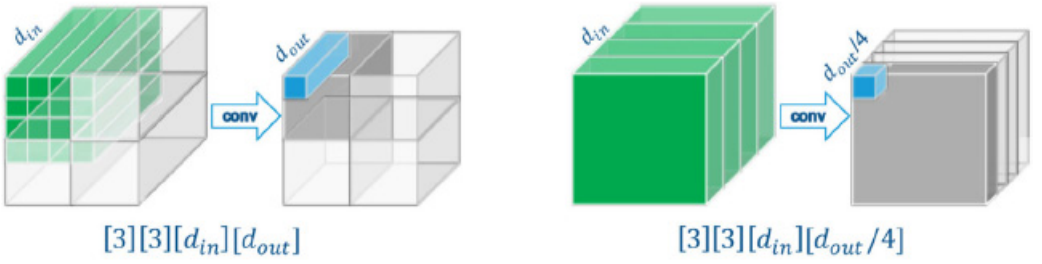


Figure 2.16. *Activation splitting across MPPA3 compute clusters*

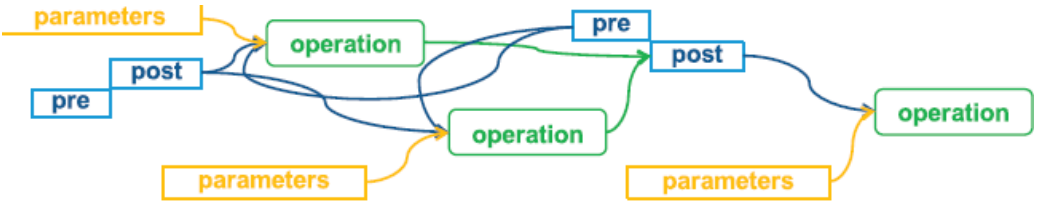


Figure 2.17. *KaNN augmented computation graph*

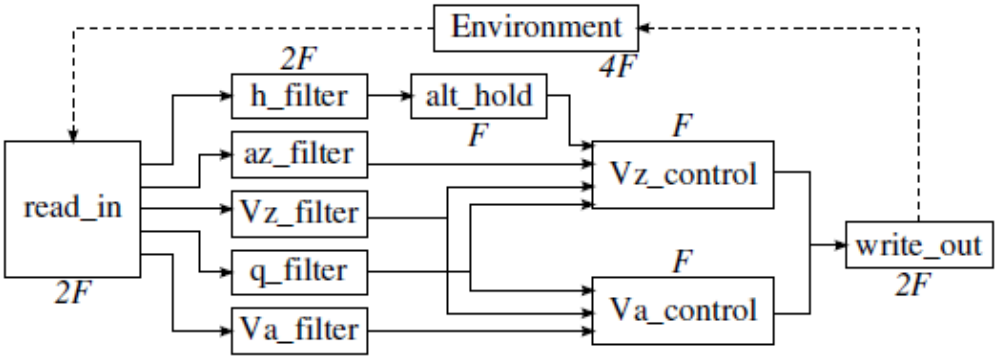


Figure 2.18. *ROSACE harmonic multi-periodic case study (Graillat et al. 2018)*

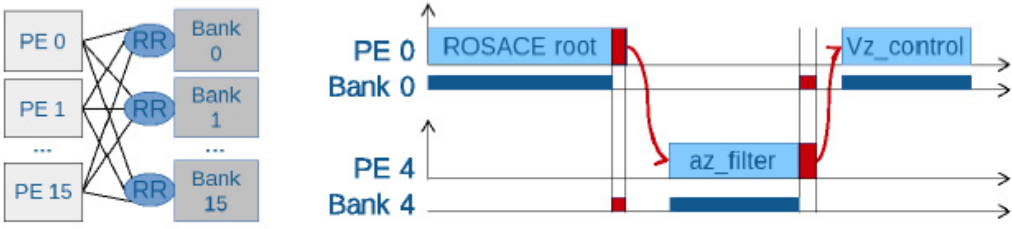


Figure 2.19. *MCG code generation of the MPPA processor*

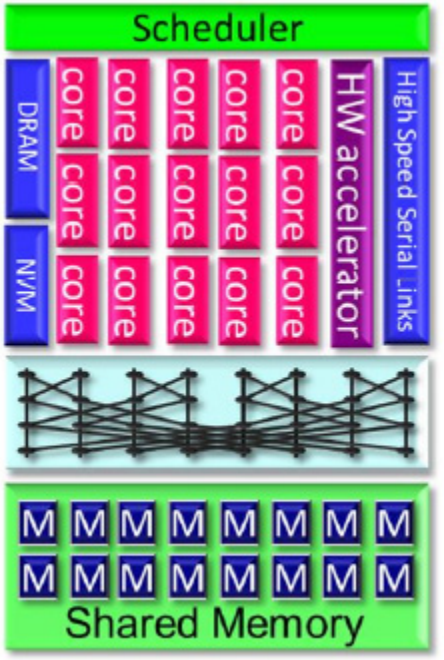


Figure 3.1. *Plural many-core architecture. Many cores, hardware accelerators and multiple DMA controllers of I/O interfaces access the multi-bank shared memory through a logarithmic network. The hardware scheduler dispatches fine grain tasks to cores, accelerators and I/O*

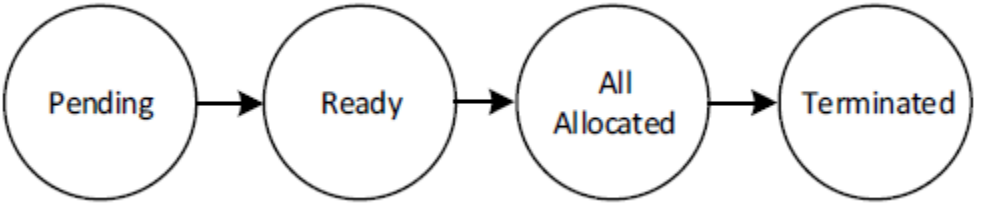


Figure 3.2. *Task state graph*

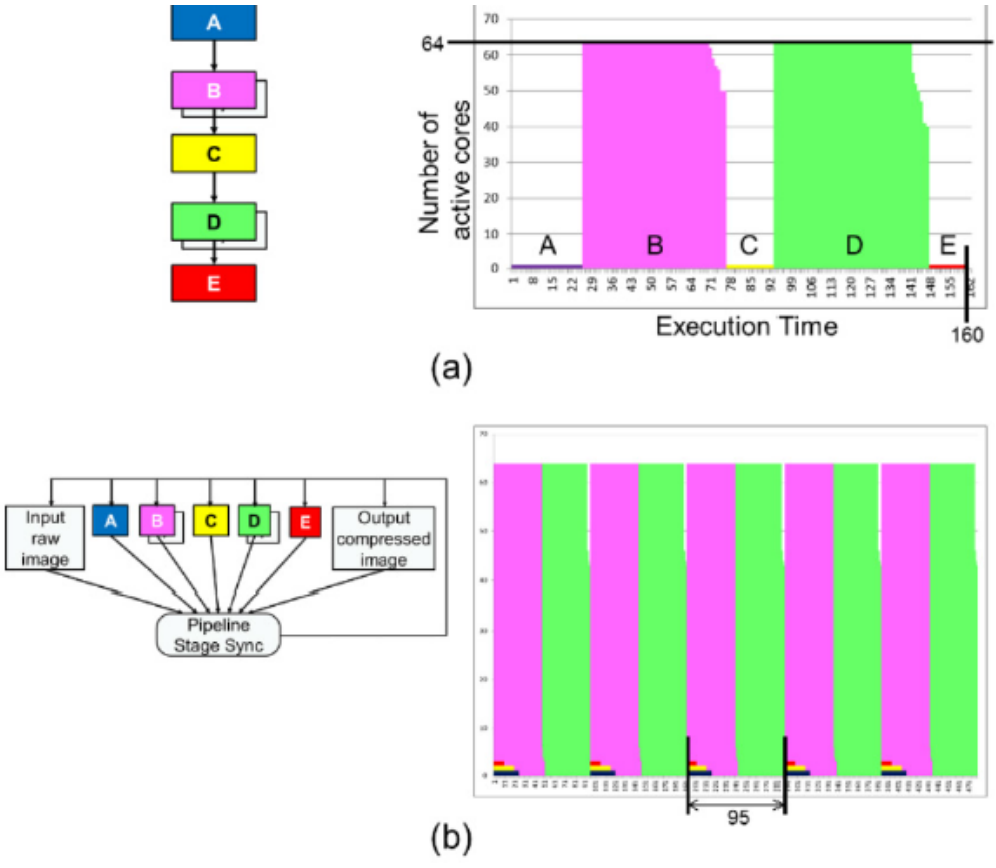


Figure 3.3. *Many-flow pipelining: (a) Task graph and single execution of an image compression program; (b) Many-flow task graph and its pipelined execution*

Core #	State	Task #	Instance #	...	...
0					
1					
2					
...					

Figure 3.4. *Core m anagement table*

Task #	Duplication quota	Dependencies	State	# Allocated instances	# Terminated instances
0					
1					
2					
...					

Data from task graph

Figure 3.5. *Task management table*

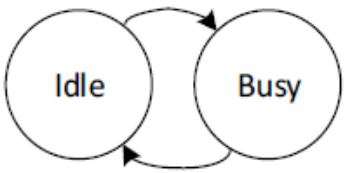


Figure 3.6. *Core state graph*

```
ALLOCATION
1. Choose a Ready task (according to priority, if specified)
2. While there is still enough scheduler capacity and there
   are still Idle cores
   a. Identify an Idle core
   b. Allocate an instance to that core
   c. Increase counter of allocated task instances
   d. If # allocated instances == quota, change task state
      to All Allocated and continue to next task (step 1)
   e. Else, continue to next instance of same task (step 2)

TERMINATION
1. Choose a core which has sent a termination message
2. While there is still enough scheduler capacity
   a. Change core state to Idle
   b. Increment # terminated instances
   c. If # terminated instances == quota, change task state
      to Terminated
   d. Recompute dependencies for all other tasks that depend
      on the terminated task, and where relevant change
      their state to Ready
```

Figure 3.7. *Allocation (top) and termination (bottom) algorithms*

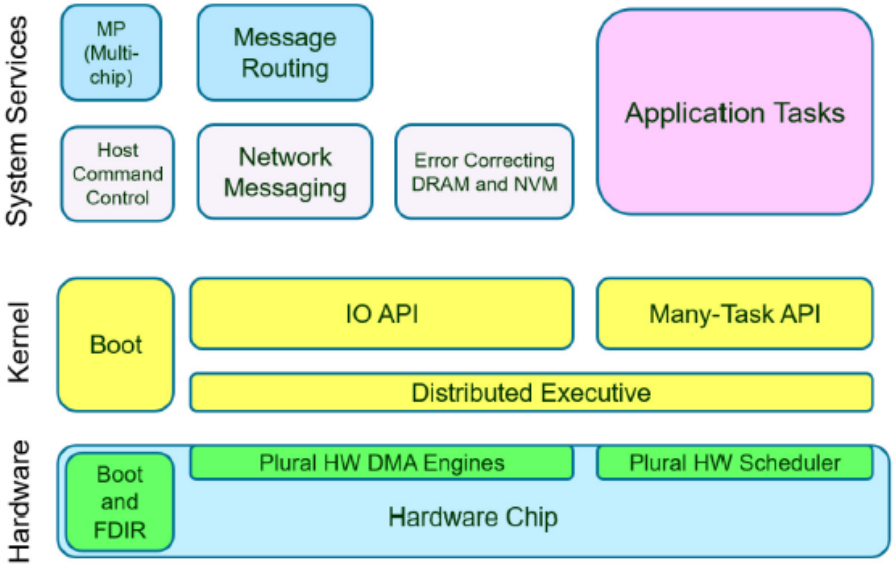


Figure 3.8. *Plural run-time software. The kernel enables boot, initialization, task processing and I/O. Other services include execution of host commands, networking and routing, error correction and management of applications distributed over multiple Plural chips*

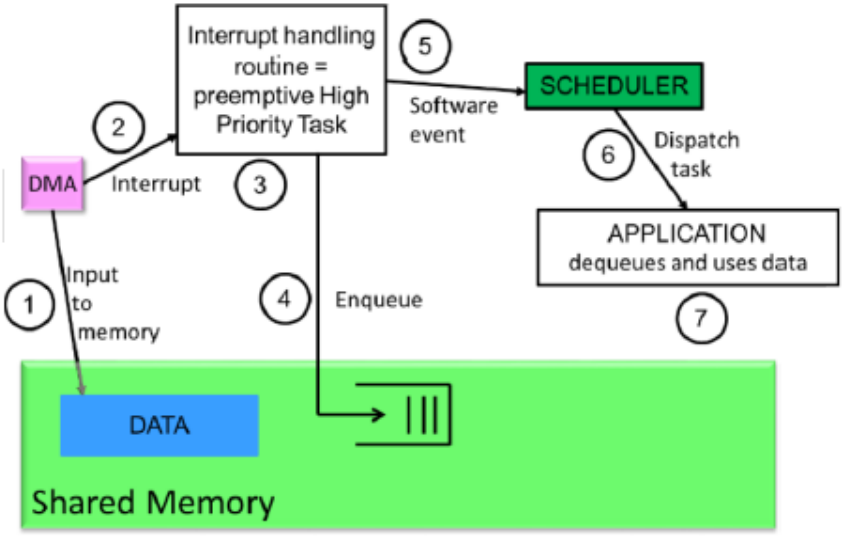


Figure 3.9. *Event sequence performing stream input*

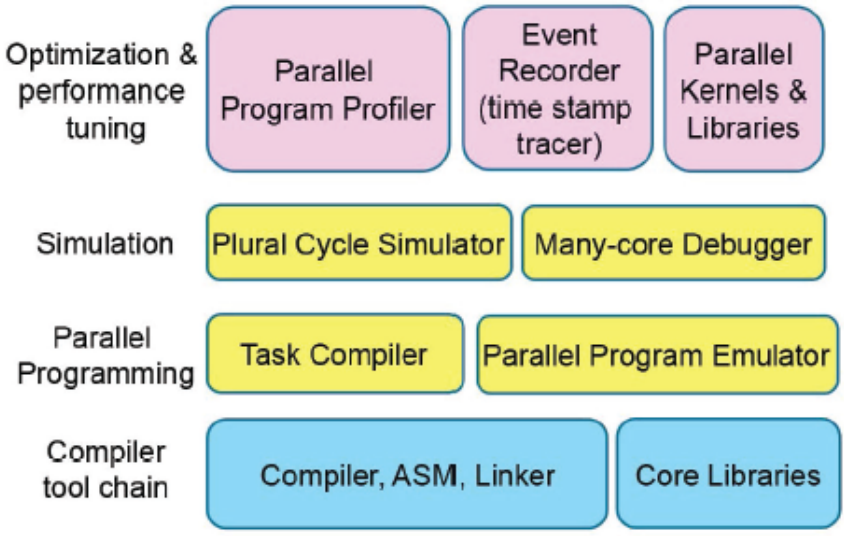


Figure 3.10. *Plural software development kit*

```
1  #define M 100
2  float A[M][M], B[M][M], C[M][M];
3
4  /* REGULAR */
5  int mm_start()
6  {
7      int i, j;
8      for (i=0; i < M; i++)
9          for(j=0; j < M; j++)
10             { A[i][j] = 13; B[i][j] = 9;}
11 }
12
13 /* DUPLICABLE */
14 void mm(unsigned int id)
15 {
16     int i, j, m;
17     float sum = 0;
18     i = id % M;
19     j = id / M;
20     for (m=0; m < M; m++)
21         sum += A[i][m] * B[m][j];
22     C[i][j] = sum;
23 }
24
25 /* REGULAR */
26 int mm_end()
27 { printf("finished mm\n"); }
```

Figure 3.11. *Matrix multiplication code on the Plural architecture*

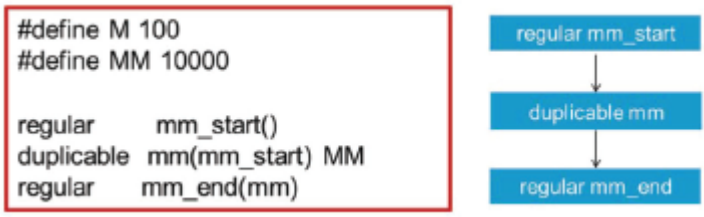


Figure 3.12. *Task graph for matrix multiplication*

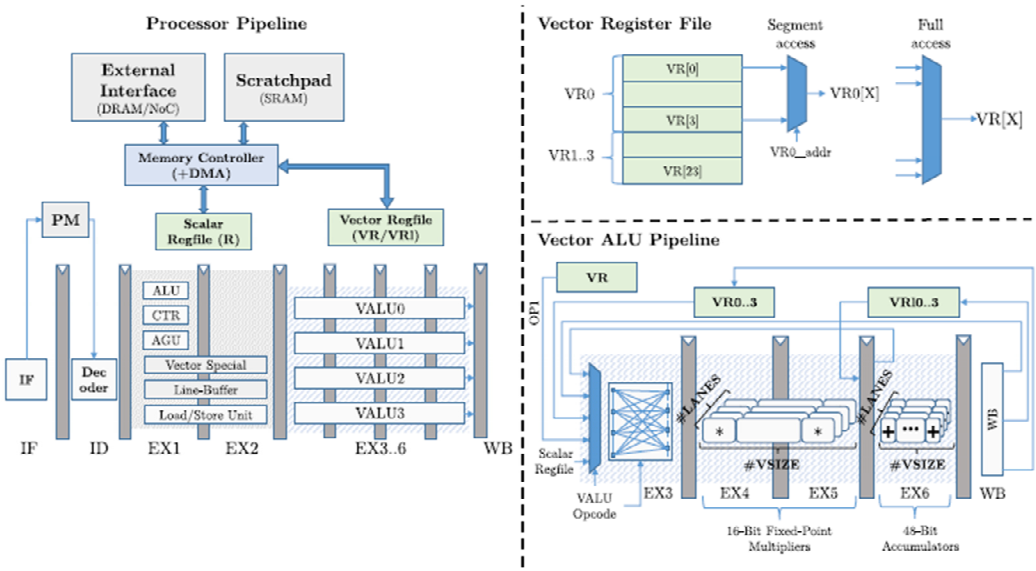


Figure 4.1. *Overview of the ASIP pipeline with its vector ALUs and register file structure*

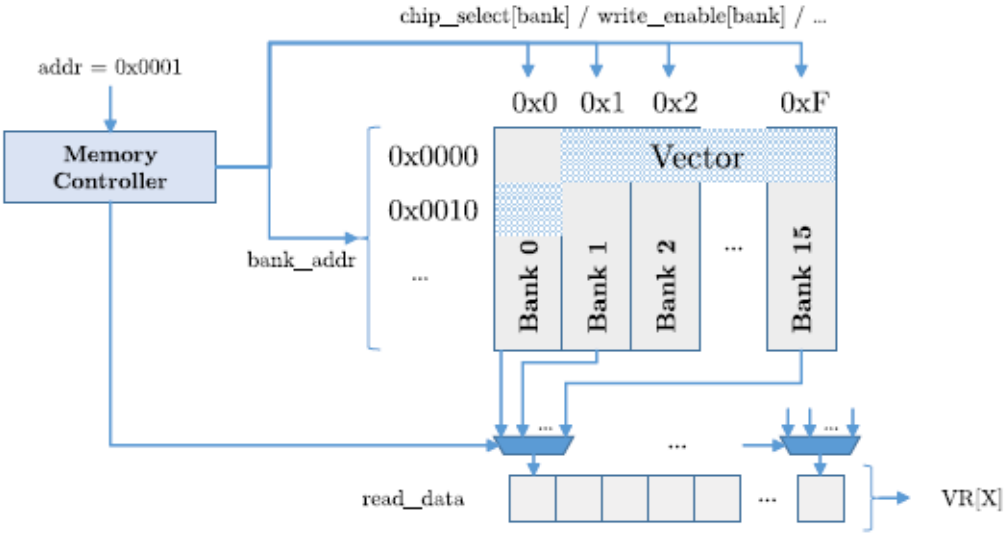


Figure 4.2. *On-chip memory subsystem with banked vector memories and an example of unaligned access at address 0x1*

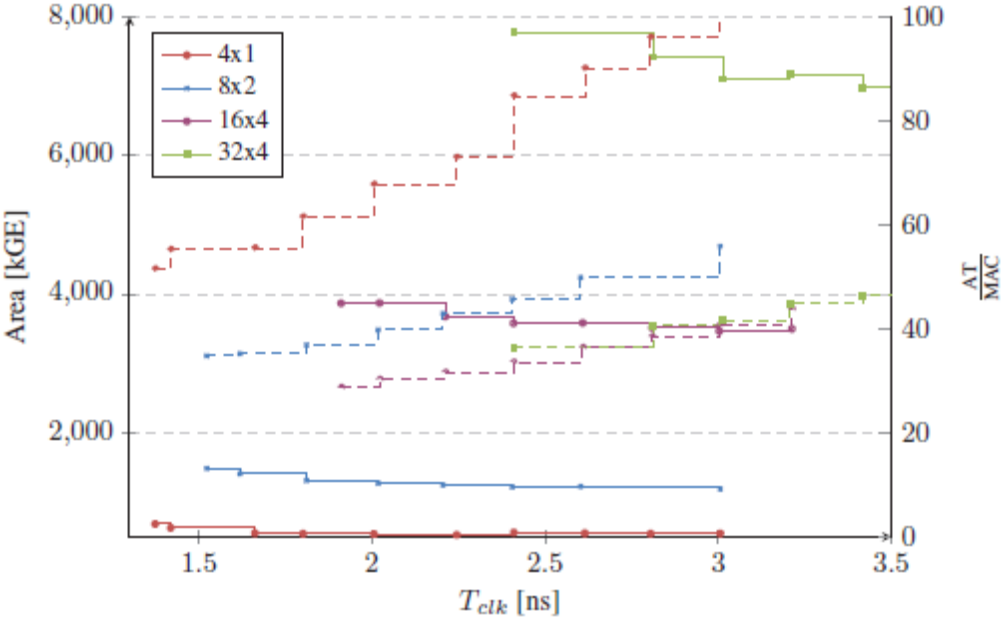


Figure 4.3. *Cell area of the synthesized cores' logic for different clock periods T_{clk} (solid lines) and area-timing product divided by the number of MAC units (dashed lines)*

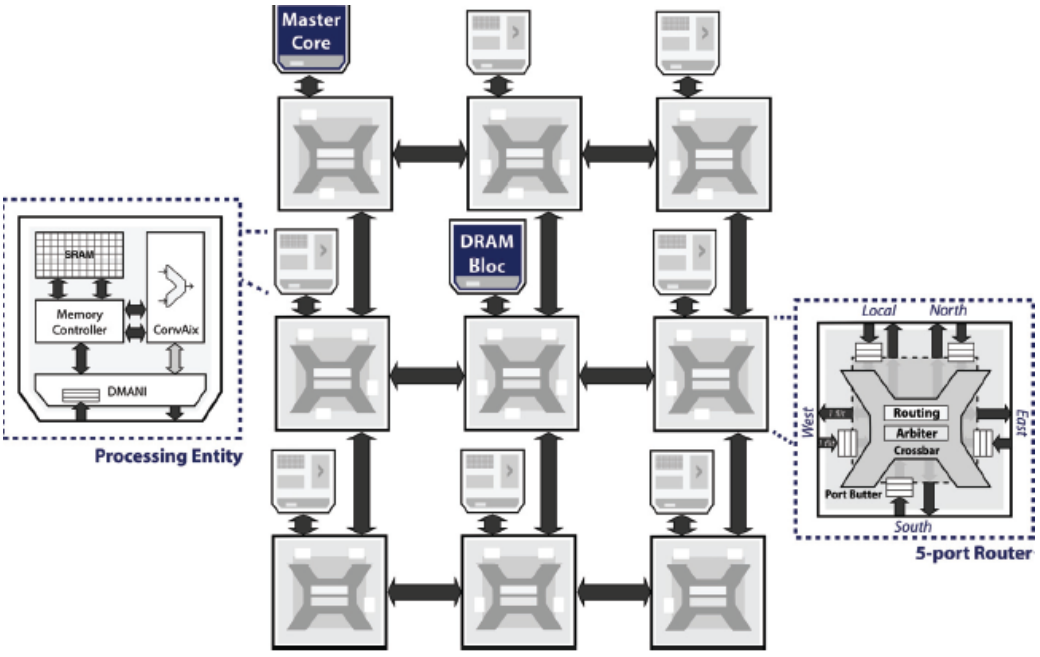


Figure 4.4. *3x3 NoC based on the HERMES framework*

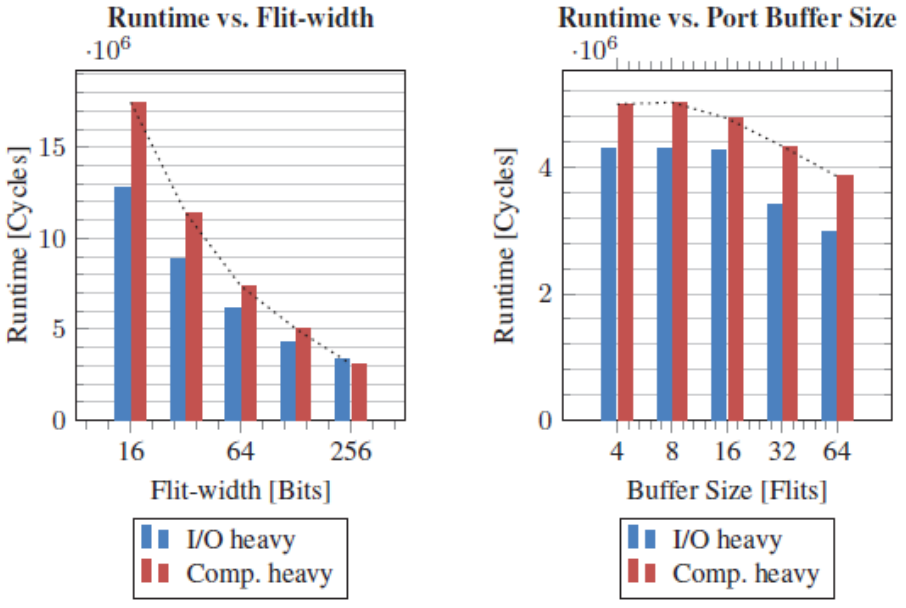


Figure 4.5. *Runtime over flit-width and port buffer size for two exemplary layers of VGG-16 that are bandwidth (I/O heavy) and compute (comp. heavy) limited*

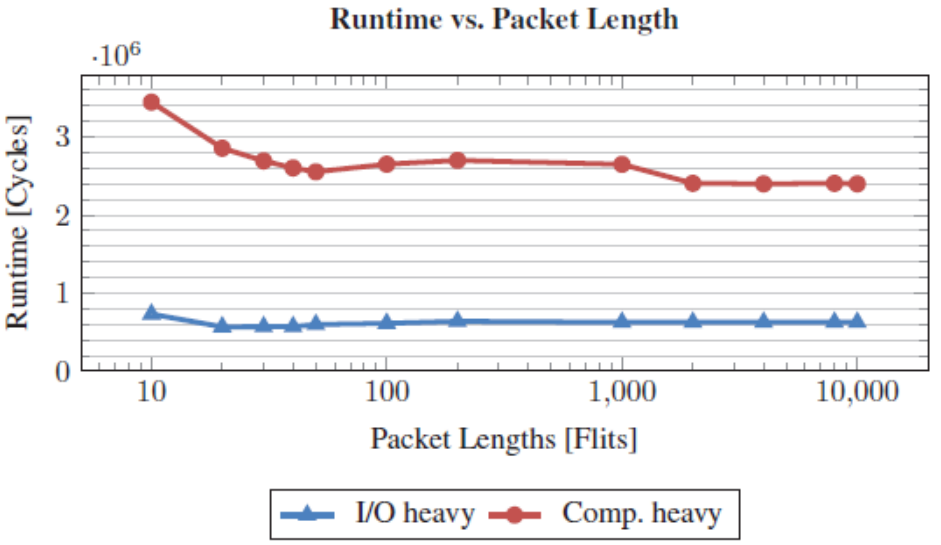


Figure 4.6. *Runtime for different packet lengths*

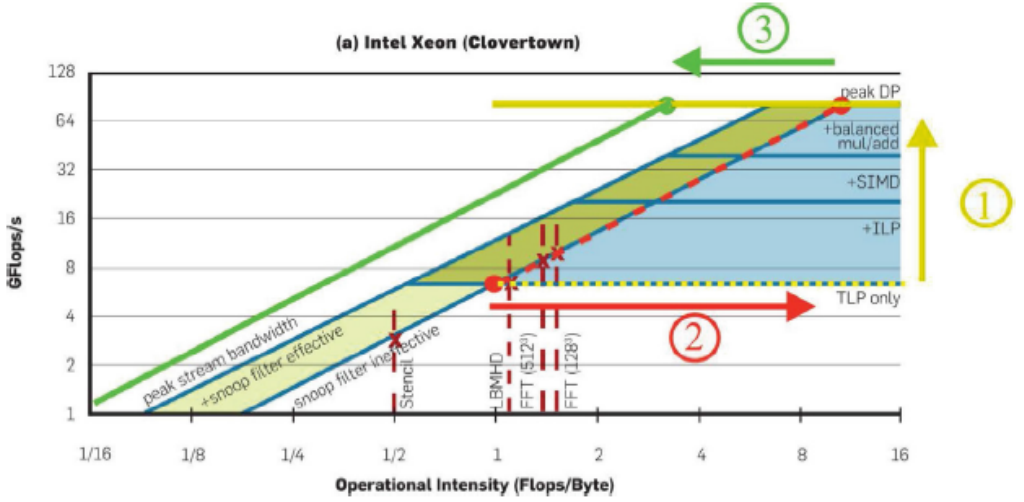


Figure 5.1. *Roofline: a visual performance model for multi-core architectures. Adapted from Williams et al. (2009)*

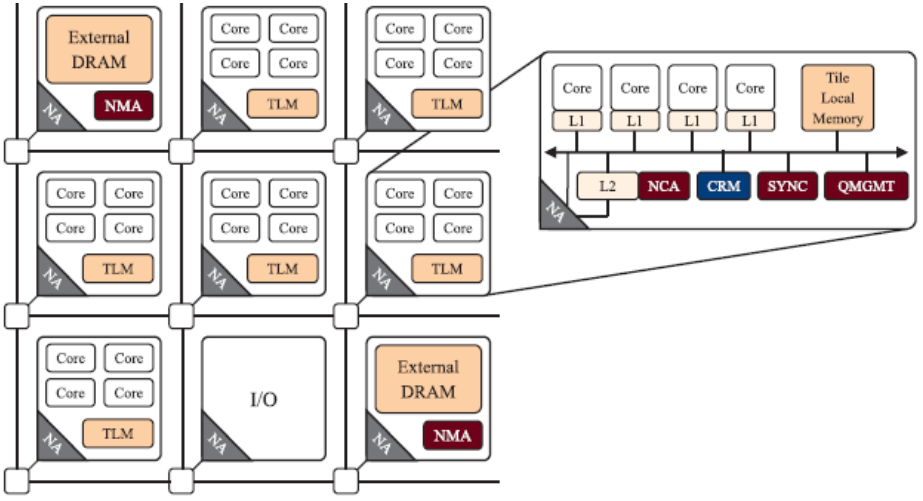
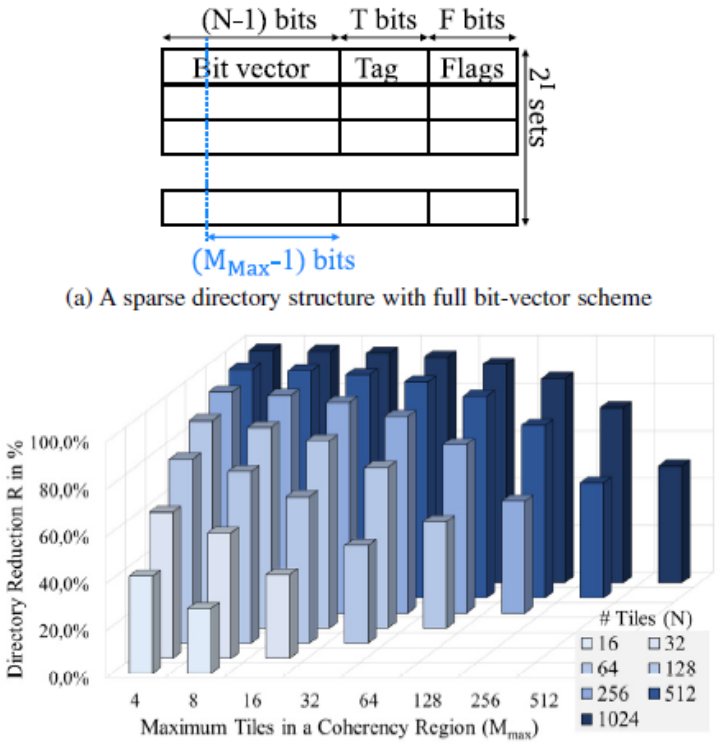


Figure 5.2. *Proposed tile-based many-core architecture*



(a) A sparse directory structure with full bit-vector scheme

(b) Plot of R for varying N and M_{max}

Figure 5.3. *Directory savings using the RBCC concept compared to global coherence*

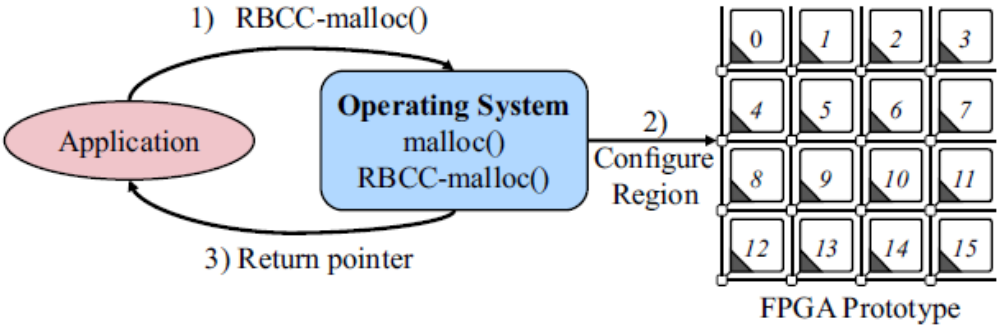


Figure 5.4. *RBCC-malloc() example*

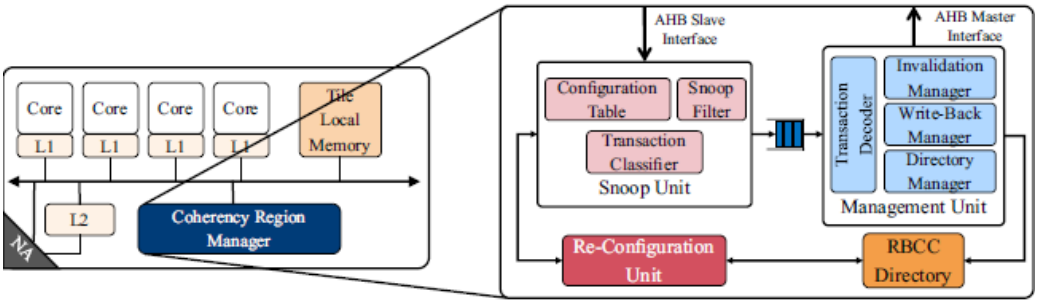


Figure 5.5. *Internal block diagram of the coherency region manager*

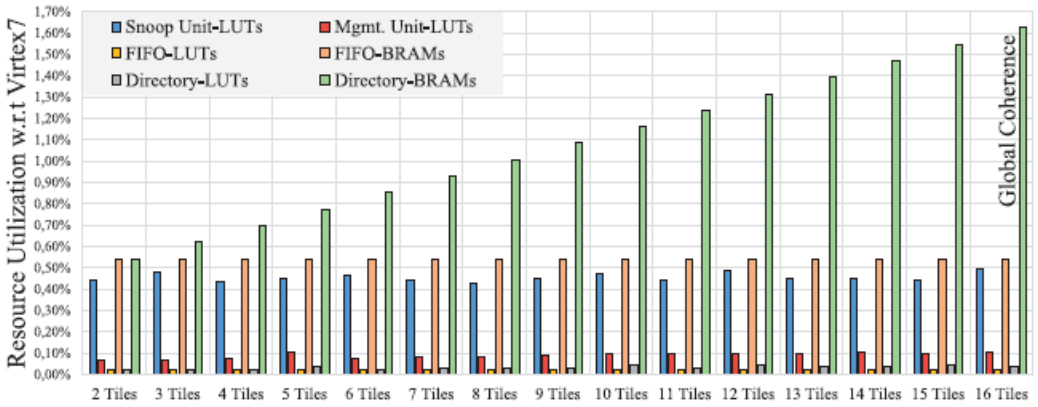


Figure 5.6. *Breakdown of the CRM's resource utilization for increasing region size*

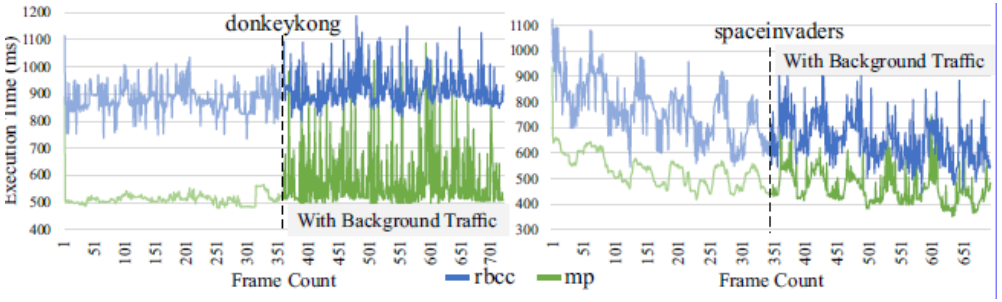


Figure 5.7. *Execution time per-frame: rbcc mode and mp mode*

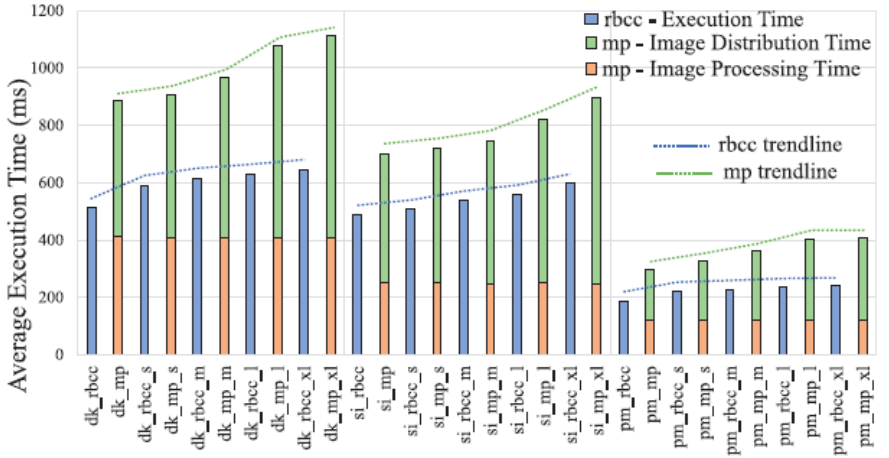


Figure 5.8. *Breakdown of execution time for different clips with increasing background traffic*

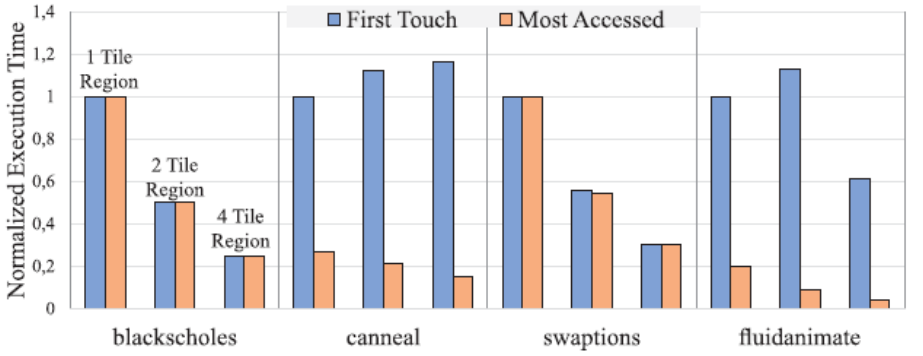


Figure 5.9. *Normalized benchmark execution time for different coherency region sizes for both data placement techniques*

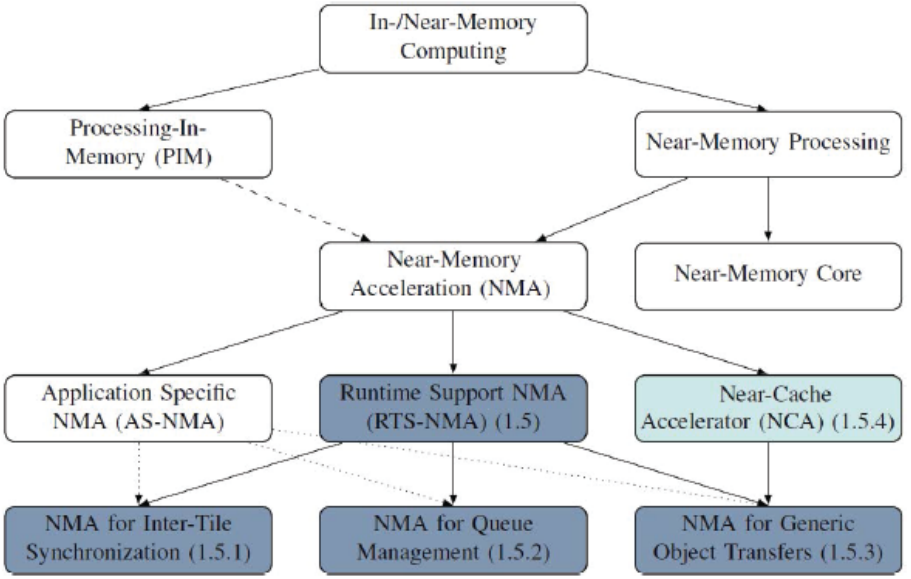


Figure 5.10. *Taxonomy of in-/near-memory computing (colored elements are covered in this section)*

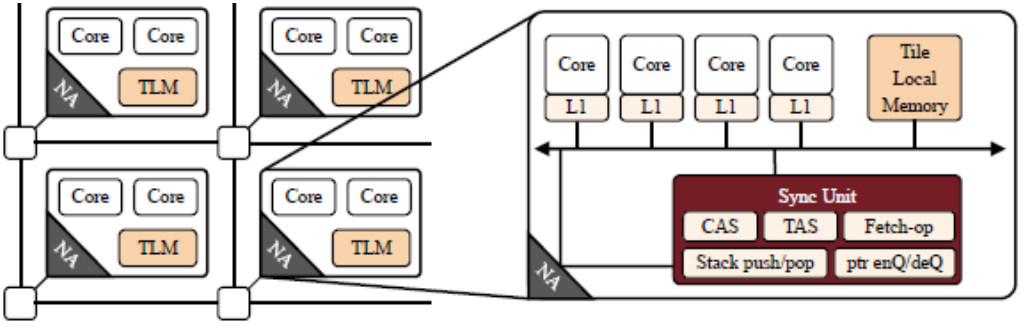


Figure 5.11. *Architecture of the remote near-memory synchronization accelerator*

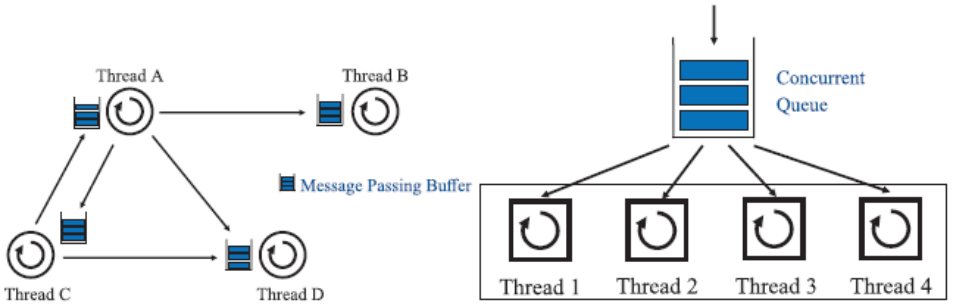


Figure 5.12. *Queues are widely used as message passing buffers*

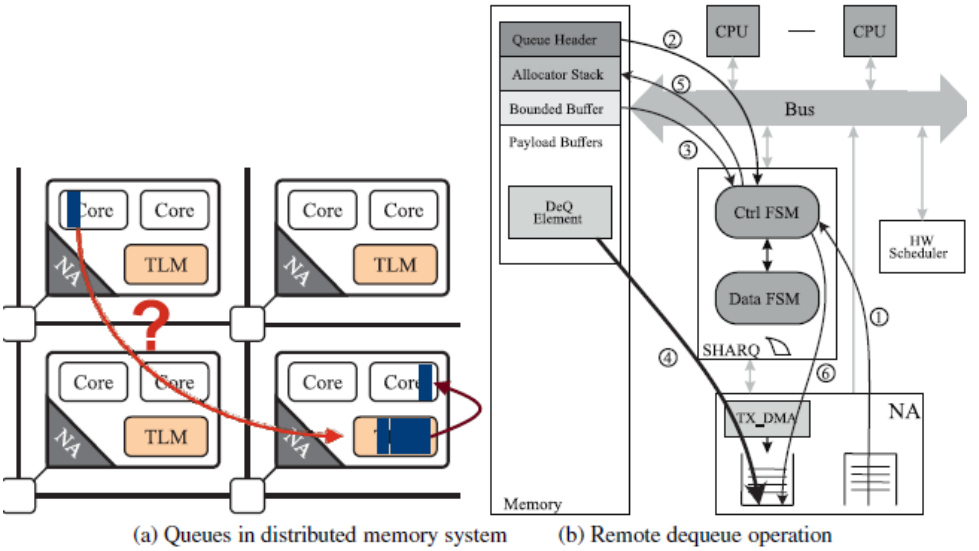


Figure 5.13. *Mechanism for a remote dequeue operation (right) for queues in distributed memory systems (left)*

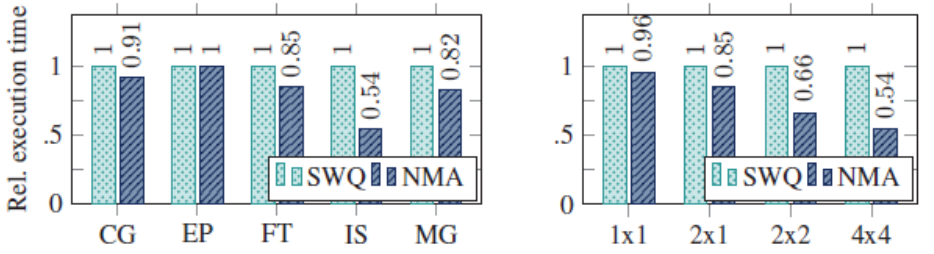


Figure 5.14. *NAS benchmark 4 × 4 results (left) and IS scalability (right) for different system sizes*

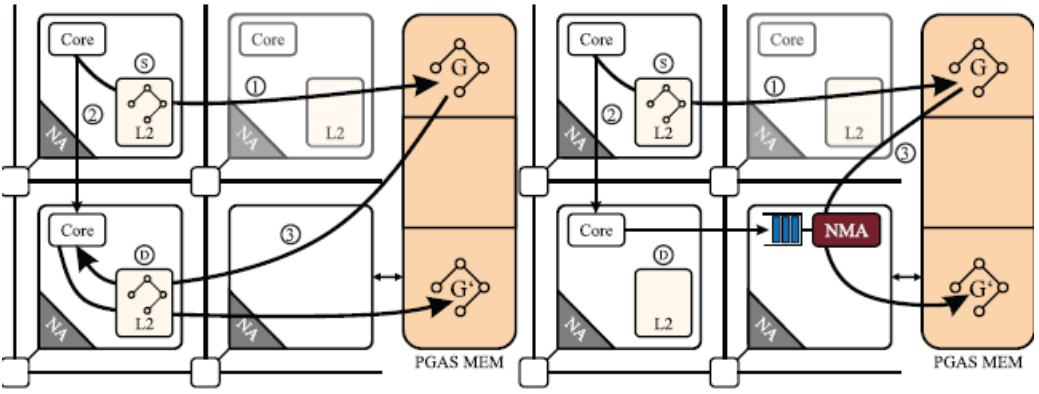


Figure 5.15. *Far-from memory (left) versus near-memory (right) graph copy*

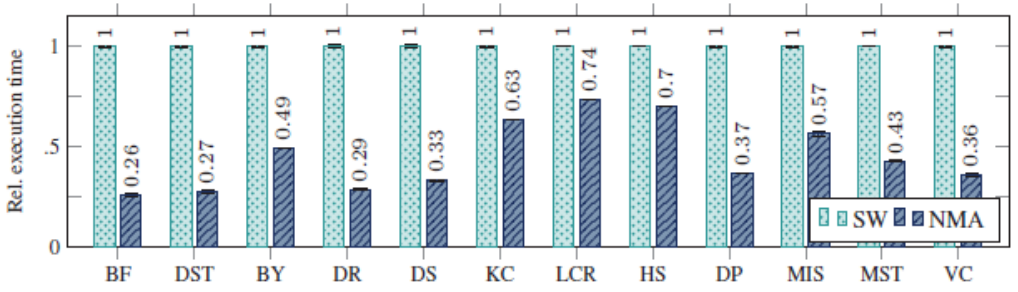


Figure 5.16. *IMSuite benchmark results on a 4 × 4 tile design with 1 memory tile*

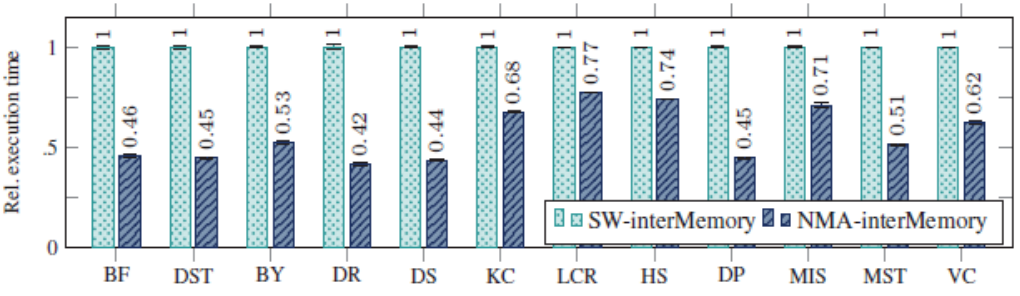


Figure 5.17. *IMSuite benchmark results for inter-memory graph copy*

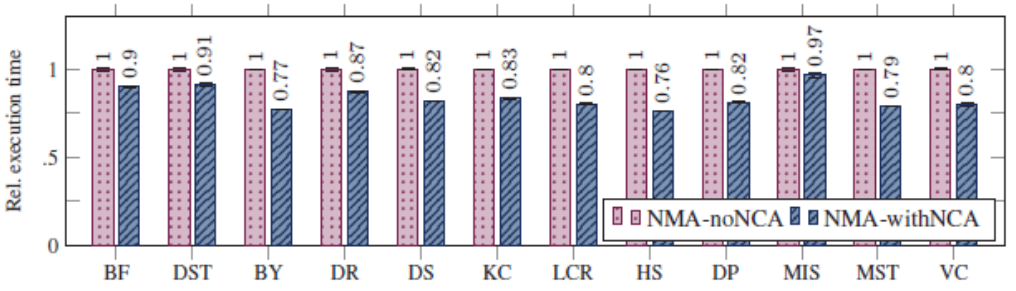


Figure 5.18. *Effect of NCA*

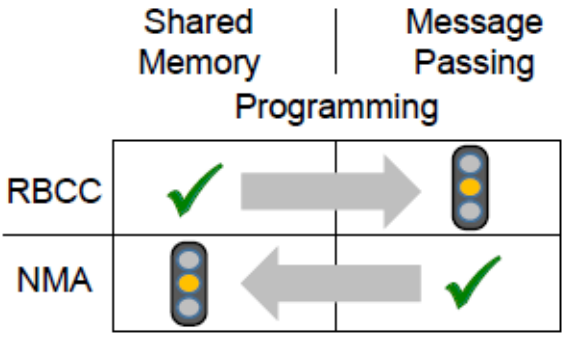


Figure 5.19. *Interplay of RBCC and NMA for shared and distributed memory programming*

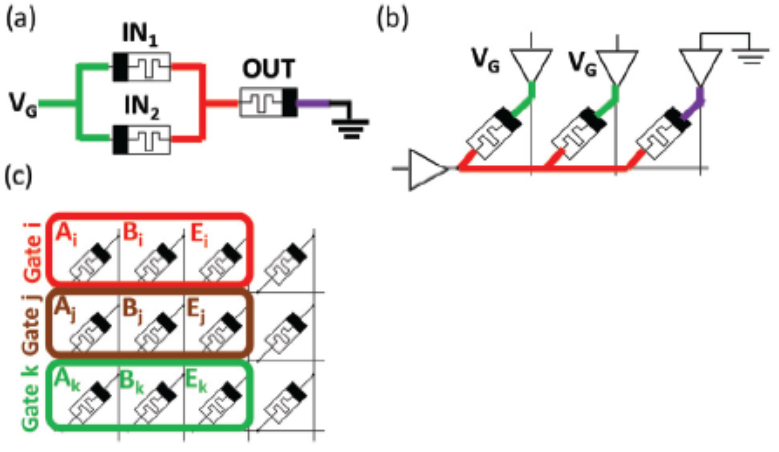


Figure 6.1. *The MAGIC NOR gate. (a) MAGIC NOR gate schematic; (b) MAGIC NOR gate within a memristive crossbar array configuration; (c) MAGIC gates operated in parallel in a memristive crossbar*

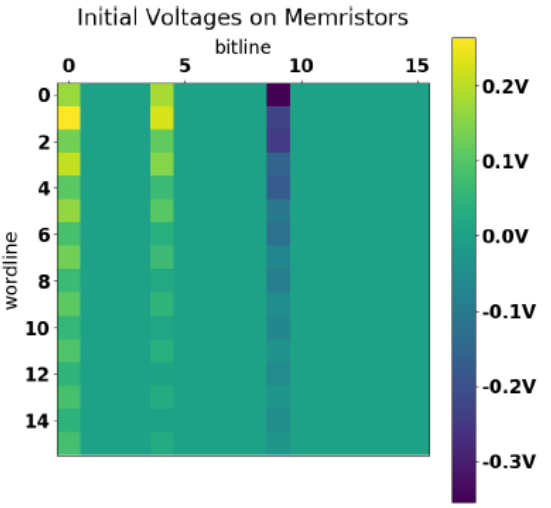


Figure 6.2. *Evaluation tool for MAGIC within crossbar arrays. The initial voltage drop on different cells in a 16 ×16 1T1R memristive crossbar array. A $V_G = 2V$ voltage was applied on columns 0 and 4, and a zero voltage was applied on column 9; other bitlinesand wordlines were floating. All the memristors were initialized with a $R_{ON} = 100\,\Omega$,except for the memristors in column 0 of the even rows, which were initialized with$R_{OFF} = 10M\,\Omega$. Negative voltages are shaded with darker colors. Negative voltage drop indicates that the memristor is changing its state to R_{OFF}*

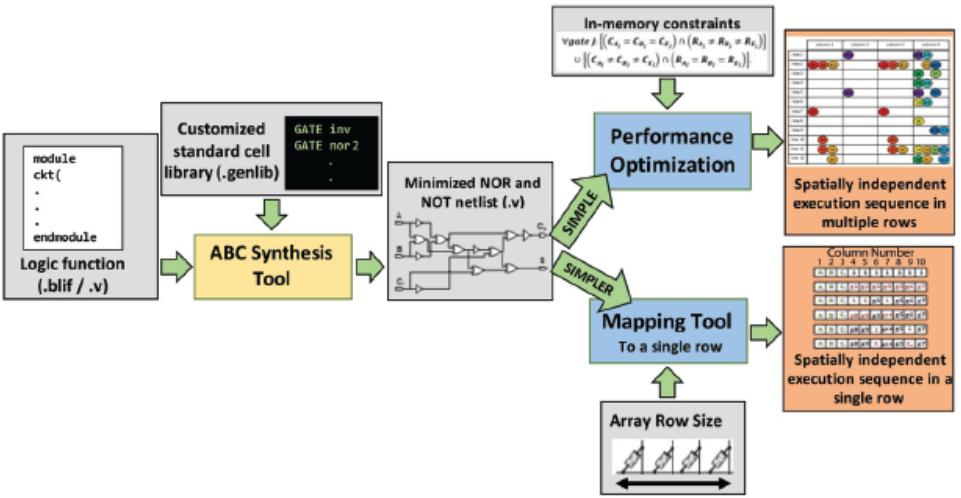


Figure 6.3. *The SIMPLE and SIMPLER flows. In both flows, the logic is synthesized to a NOT and NOR netlist, using the ABC tool. Then, the flows are split into different mapping methods. The SIMPLE flow optimizes the execution latency and maps onto several rows in the memristive array. The SIMPLER flow optimizes throughput and maps onto a single row in the memristive array*

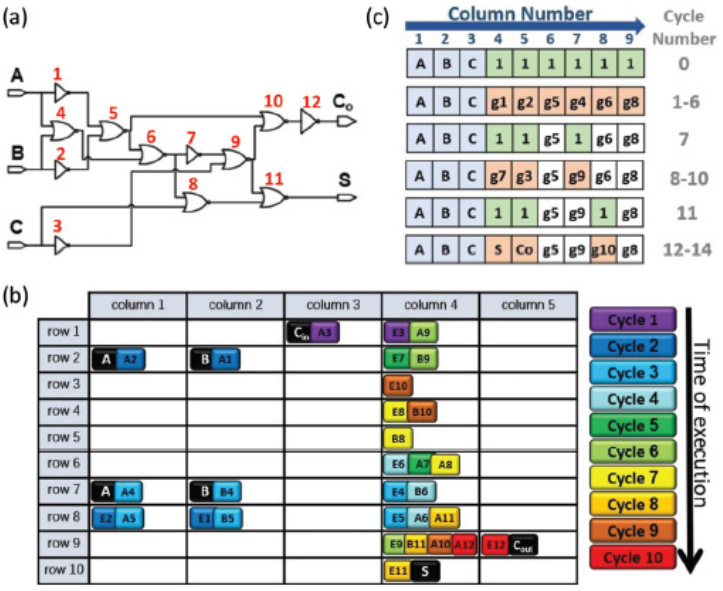


Figure 6.4. *A 1-bit full adder implementation using SIMPLER. (a) A 1-bit full adder NOT and NOR netlist, generated by the ABC tool. (b) SIMPLE execution sequence. Each clock cycle is shown in a different color. The logic gates that operate in a clock cycle are shown with the clock cycle color. (c) SIMPLER execution sequence. Green cells are initialized, and orange cells are written with a new gate output value*

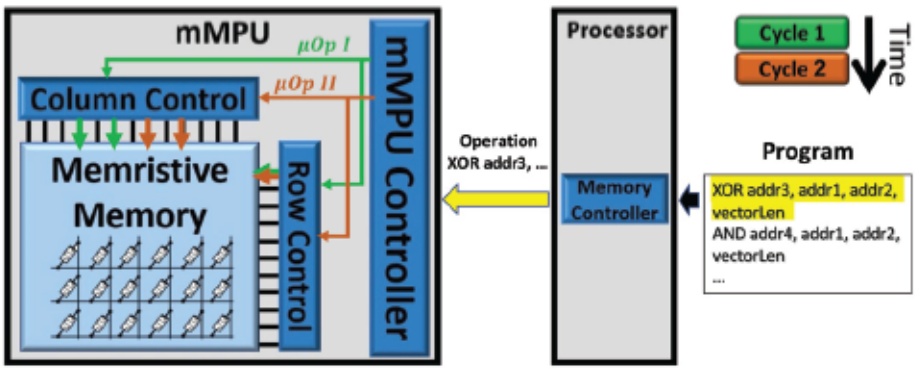


Figure 6.5. *High-level description of the mMPU architecture. A program is executed by the processor. The memory controller within the processor sends a command to the mMPU controller (yellow arrow), which splits the command into micro-operations (MAGIC NOR execution sequence). Then, in each clock cycle, the mMPU controller executes a micro-operation by applying appropriate voltages on the bitlines and wordlines of the memristive crossbar array (green and brown arrows). The command vector length field indicates the number of wordlines participating in the command calculation and operating in parallel*

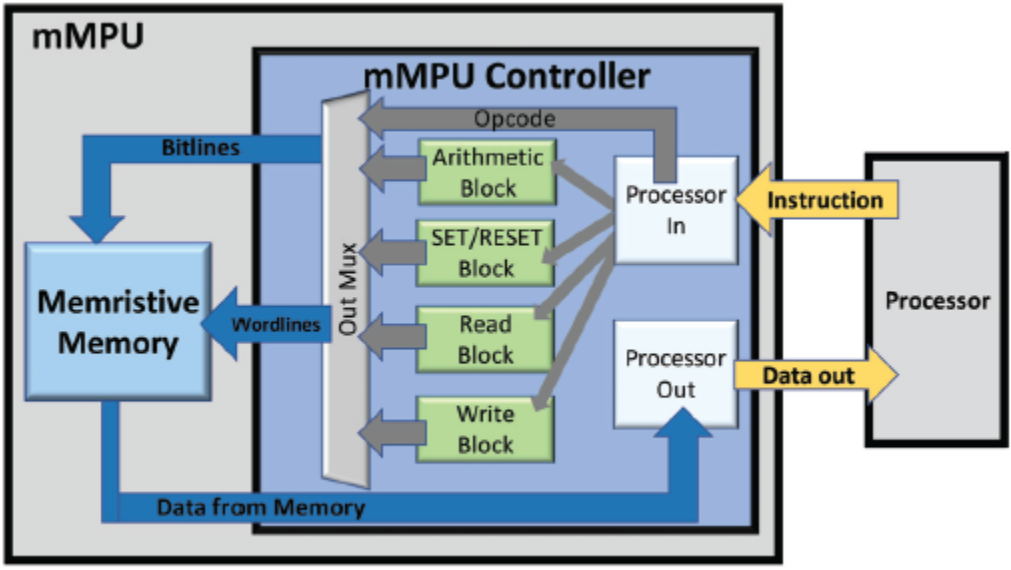


Figure 6.6. *The internal structure of the mMPU controller. First, an instruction is sent from the processor to the mMPU controller and interpreted in the processor in block. Then, it is sent for further interpretation in a suitable block (arithmetic, SET/RESET/read or write), according to the instruction type. The output of these blocks are the control signals, which are propagated to the memristive array bitlines and wordlines. The output is then transferred to the processor out unit, which sends it back to the processor*

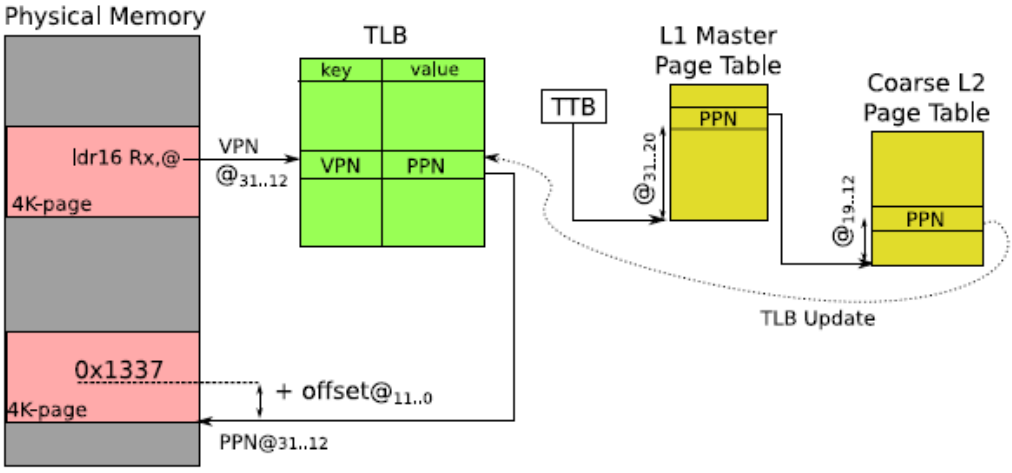


Figure 7.1. *Address translation for the ARMv7 architecture*

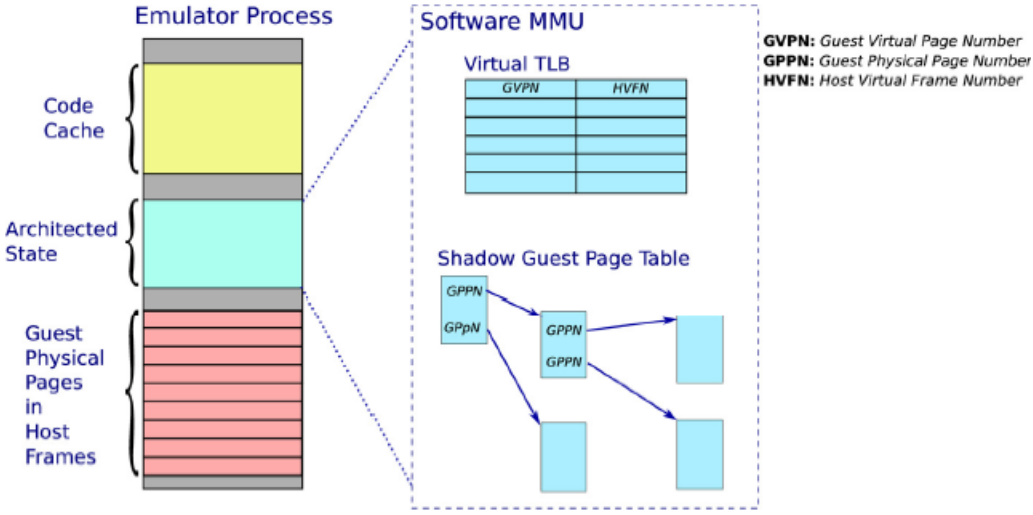


Figure 7.2. *Host view of the architected state of the guest*

```
uint32_t ldr_helper(uint32_t addr) {
    uint32_t hvpn;
    uint32_t gppn;
    // extract the guest virtual page number and offset
    uint32_t gvpn = guest_virtual_page_number(addr);
    uint32_t offset = guest_virtual_offset(addr);
    // access the virtual TLB, update if necessary
    uint32 index = hash(gvpn);
    if (virtual_tlb[index].key != gvpn) {
        gppn = shadow_page_table_walk(cpu->ttb, gvpn);
        hvfn = guest_physical_to_host_virtual(gppn);
        HT[index].entry = hvfn;
    } else
        hvfn = HT[index].entry;
    // do the guest memory access, reading the 32-bit value
    return *(uint32_t*)(hvfn + offset);
}
```

Figure 7.3. *Pseudo-code of the helper for the ldr instruction*

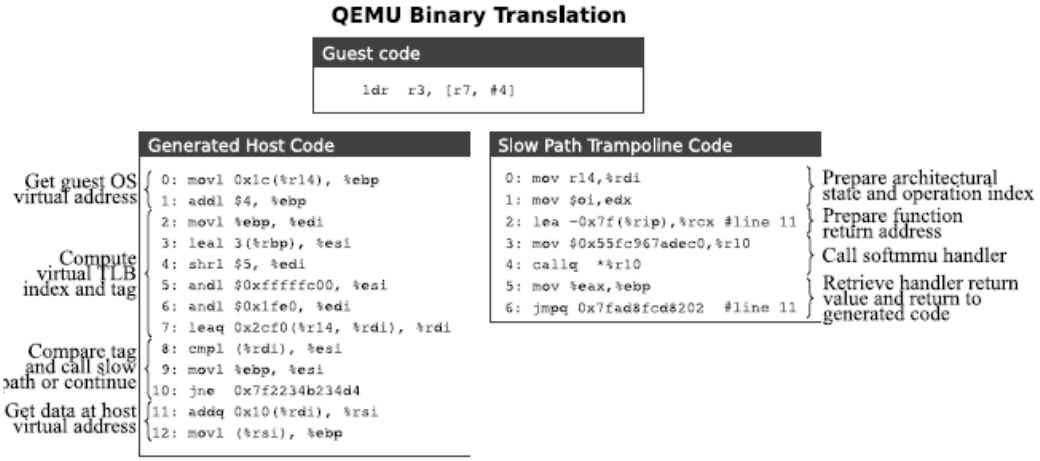


Figure 7.4. *QEMU-generated code to perform a load instruction*

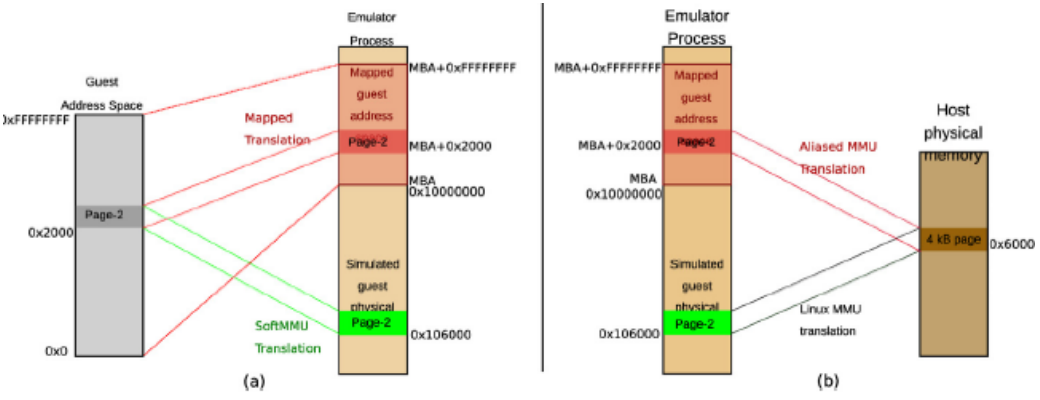


Figure 7.5. *Embedding guest address space*

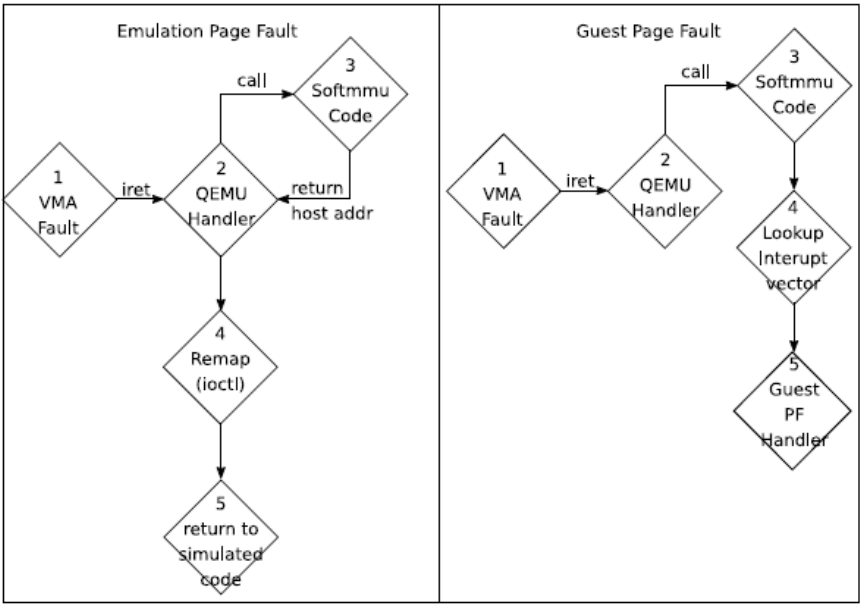


Figure 7.6. *Overview of the implementation*

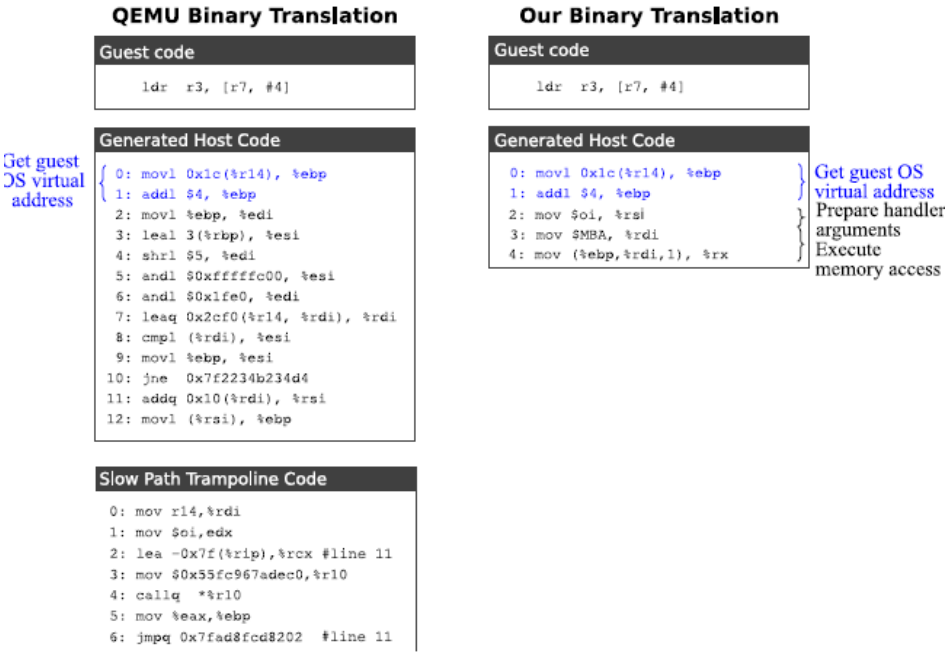


Figure 7.7. *Contrasting memory access binary translations*

```
Data: pf: page fault information, including register values
Result: update embedded guest mapping or forward fault to simulator
if isReadAccess() then
    guest_addr ← pf.addr − MBA
    guest_phys_addr ← guest_page_table(guest_addr)
    if is_valid(guest_phys_addr) then
        host_virtual_frame ← qemu_frame_table(guest_phys_addr)
        if host_virtual_frame != NULL then
            setup_host_mmu_alias(pf.addr, host_virtual_frame)
            return
        end
    end
end
pf.reg[pc] ← qemu_handler
return
```

Figure 7.8. *Kernel module page fault handler*

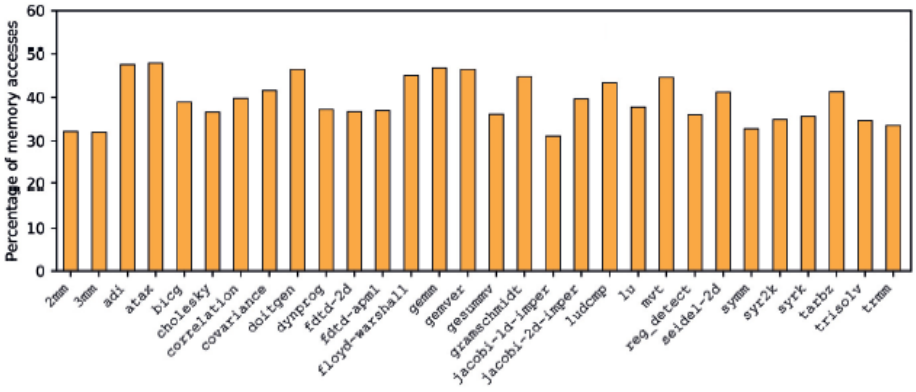


Figure 7.9. *Percentage of memory accesses (with Linux)*

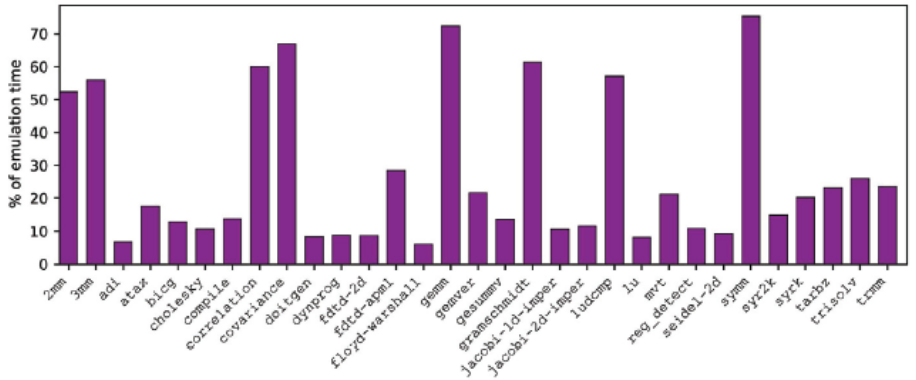


Figure 7.10. *Time spent in the Soft MMU*

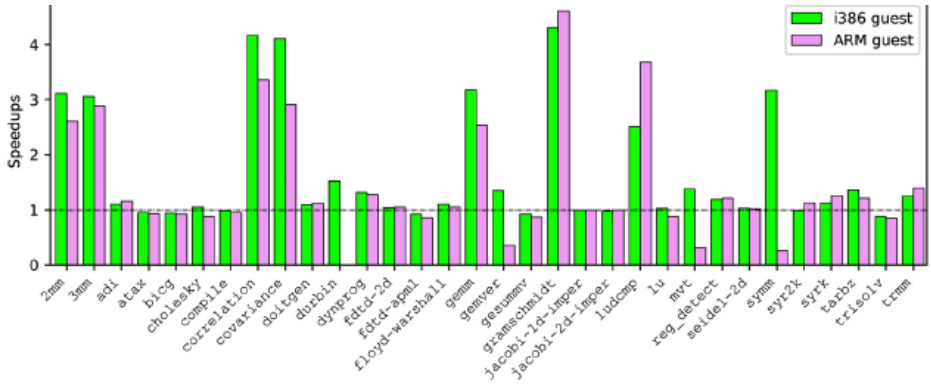


Figure 7.11. *Benchmark speed-ups: our solution versus vanilla QEMU*

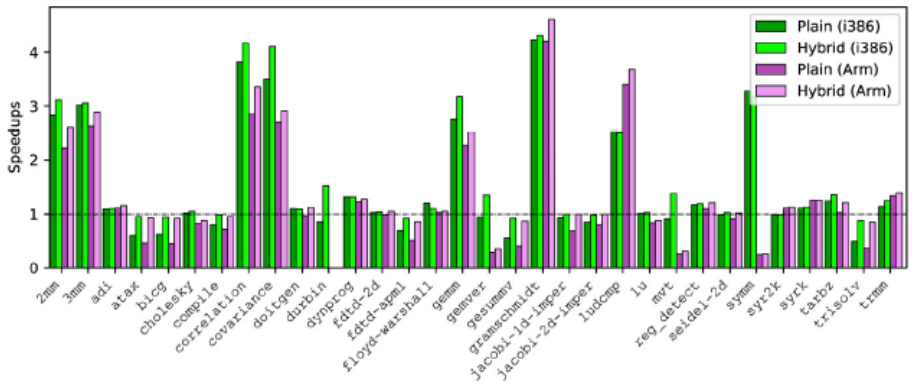


Figure 7.12. *Plain/hybrid speed-ups versus vanilla QEMU*

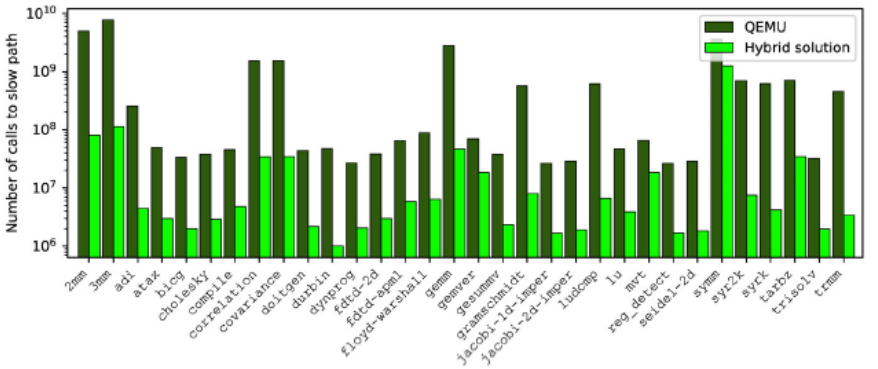


Figure 7.13. *Number of calls to slow path during program execution (i386 and ARM summed)*

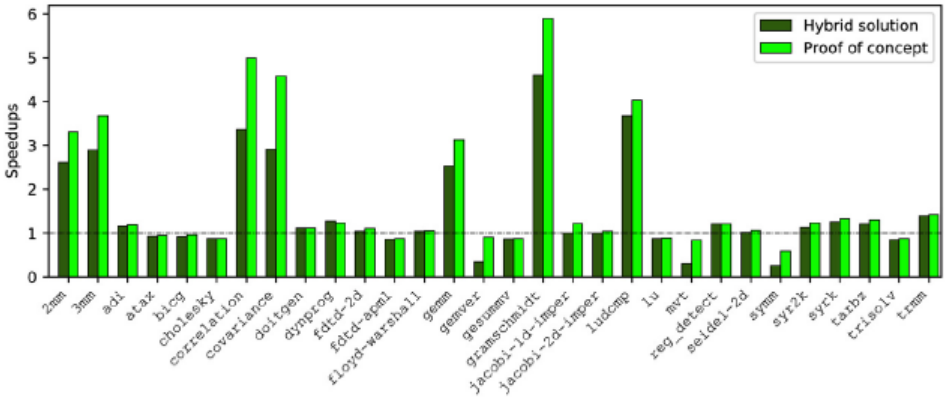


Figure 7.14. *Page fault optimization speed-ups (ARM guest)*

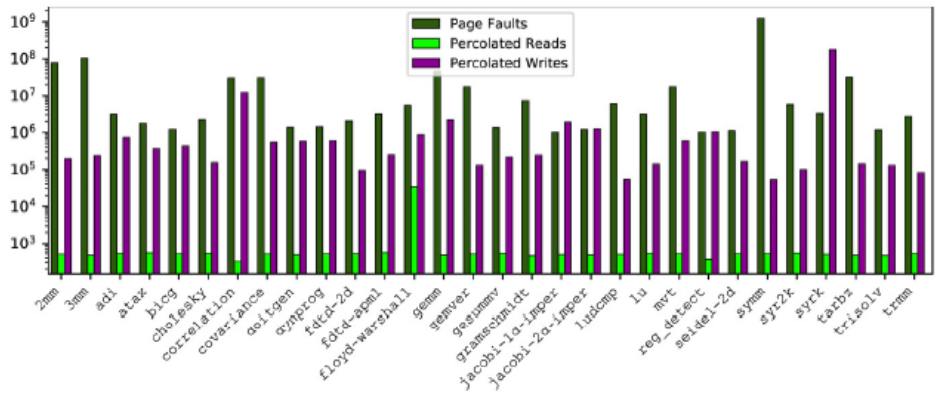


Figure 7.15. *Page fault handling – internal versus percolated (note the logarithmic scale on the y-axis)*

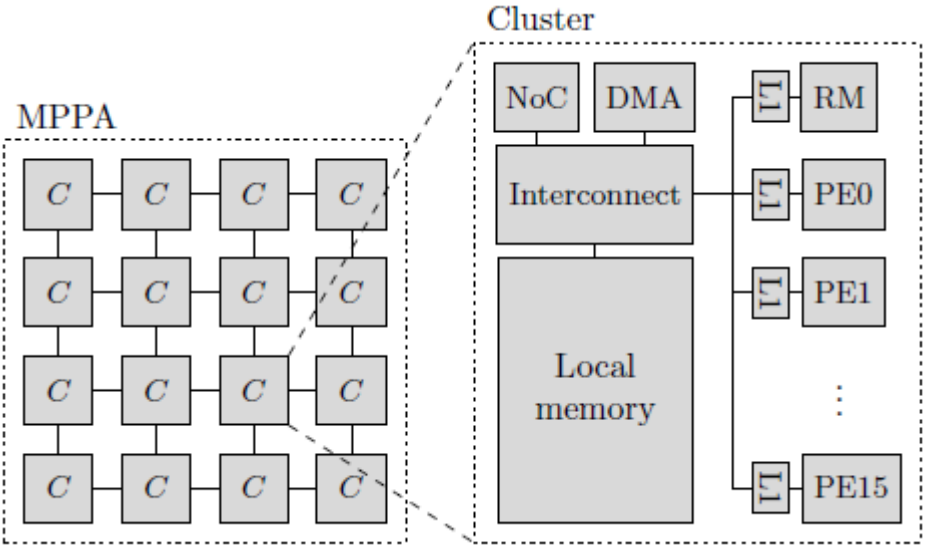


Figure 8.1. *Kalray MPPA overall architecture*

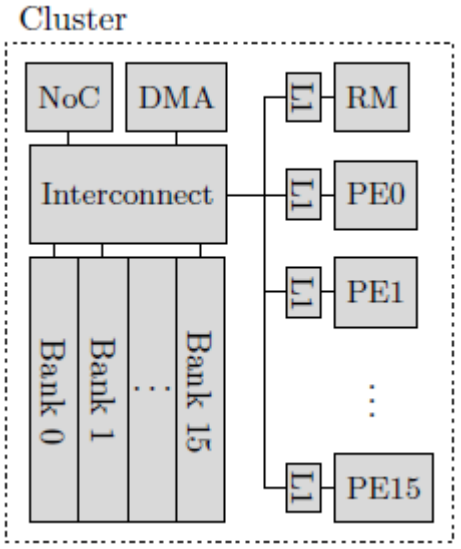


Figure 8.2. *Memory banks and local interconnect in a Kalray cluster*

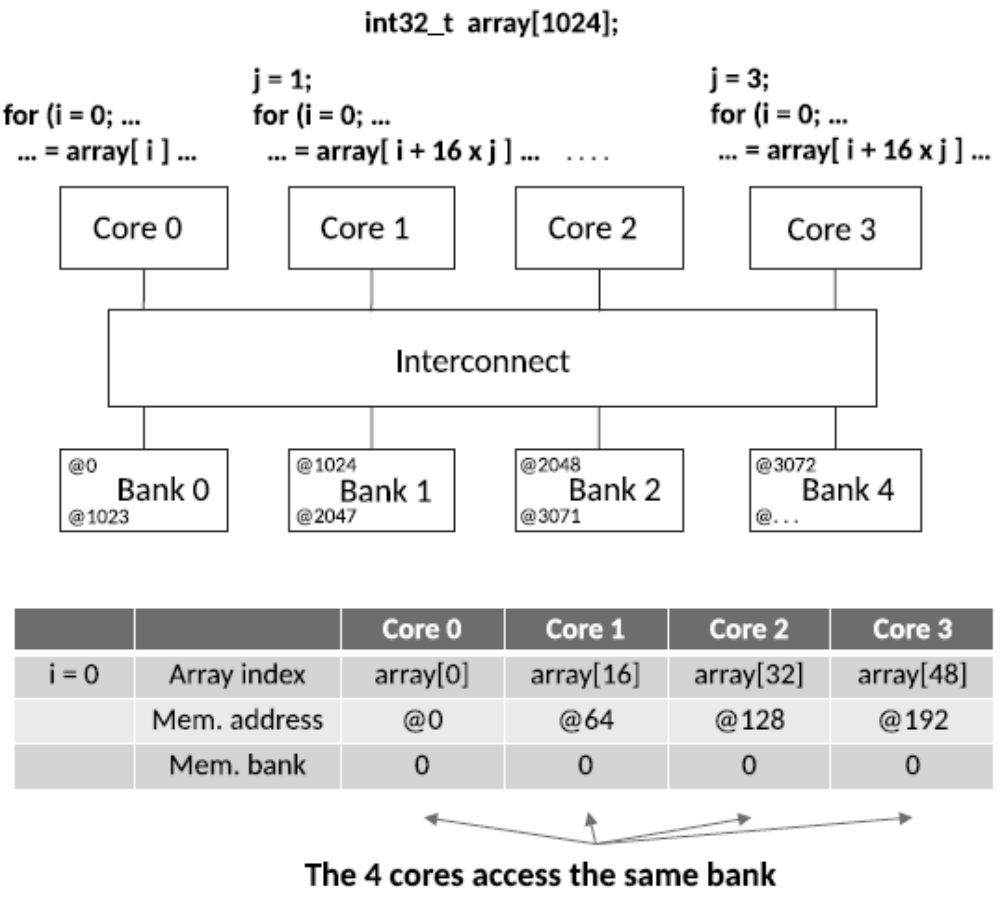


Figure 8.3. *Example of collisions with four accesses to the same memory bank*

Figure 8.4. *Description of memory address bits and their usefor a 32-bit memory with four banks of 1 kB each*

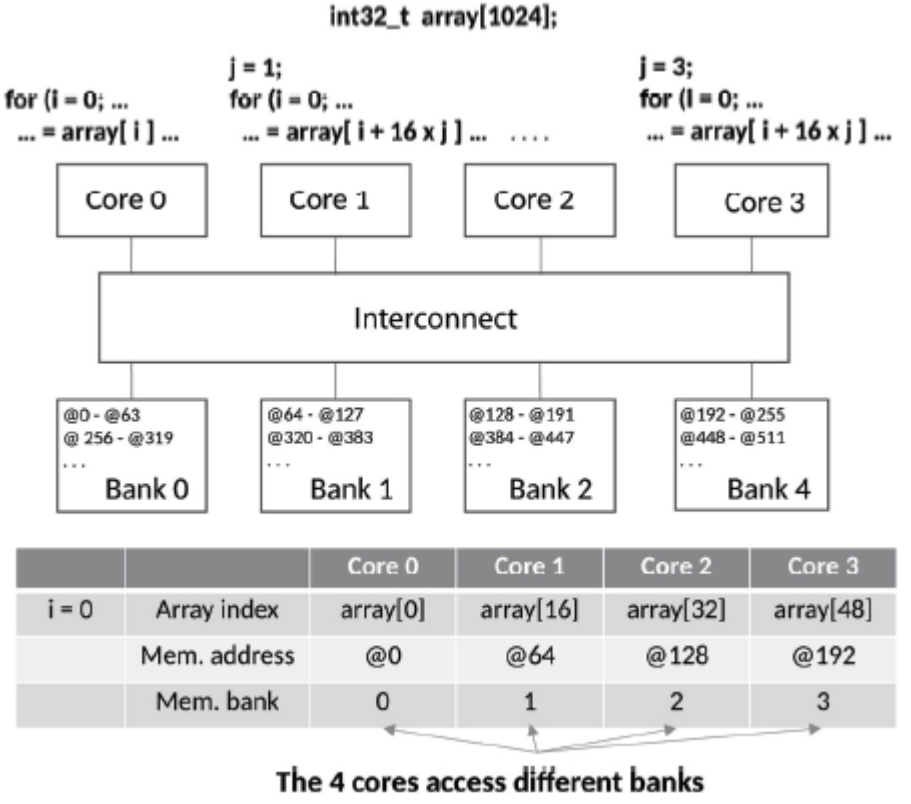


Figure 8.5. *Example of interleaving with four accesses to different memory banks*

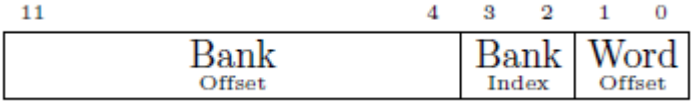


Figure 8.6. *Description of memory address bits and their use for a 32-bit memory with four banks of 1 kB each with interleaving*

		Bank			
		0	1	2	3
Address	0	•			
	16	•			
	32	•			
	48	•			
	total	4	0	0	0

Figure 8.7. *A four-bank architecture, 1-byte words, with a 16-byte stride access pattern*

		Bank			
		0	1	2	3
Address	0	•			
	17		•		
	34			•	
	51				•
	total	1	1	1	1

Figure 8.8. *A four-bank architecture, 1-byte words, with a 17-byte stride*

Bank				
0	1	2	3	4
0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19
		:		

		Bank				
		0	1	2	3	4
Address	0	•				
	16		•			
	32			•		
	48				•	
	total	1	1	1	1	0

Figure 8.9. *Left: the distribution of addresses within a 5-bank memory system. Right: a 16-byte stride access pattern in this memory system*

	Bank			
0	1	2	3	4
0	16	12	8	4
5	1	17	13	9
10	6	2	18	14
15	11	7	3	19

Figure 8.10. *Distribution of addresses across five memory banks with Index = _Addr ÷ $S_{interleave}$_ mod 2_2*

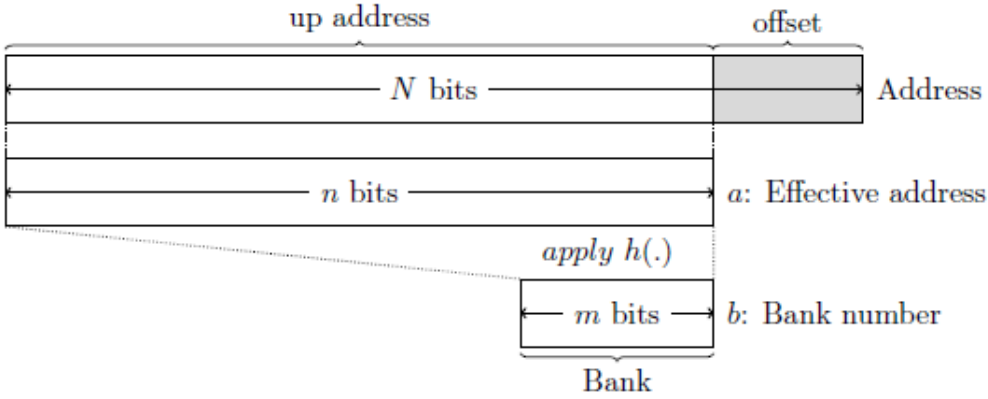


Figure 8.11. *Using a hash function for memory bank selection. N is the address size, n is the address size without the offset byte, $m$is the memory bank number size and h is the hash function used*

$$H = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix}$$

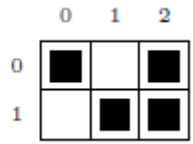


Figure 8.12. *Left: an example of an H matrix of size 2×3. Right: the same H matrix displayed as a grid*

Bank			
0	1	2	3
0	1	2	3
7	6	5	4
9	8	11	10
14	15	12	13
	:		

		Bank			
		0	1	2	3
Address	0	•			
	4				•
	8		•		
	12			•	
	total	1	1	1	1

Figure 8.13. *Example of PRIM allocation in a four-bank architecture, and four memory accesses with a 4-byte stride access pattern*

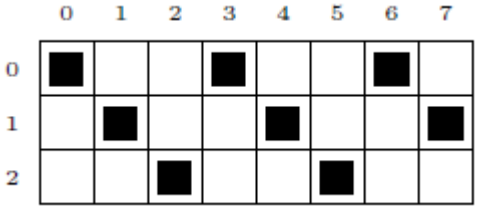


Figure 8.14. H *matrix for the PRIM solution*

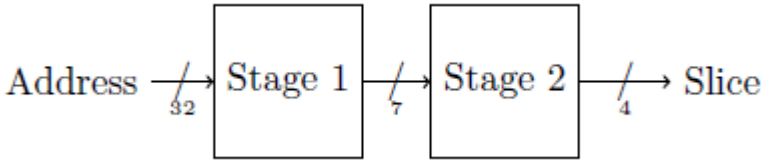


Figure 8.15. *Complex Addressing circuit overview*

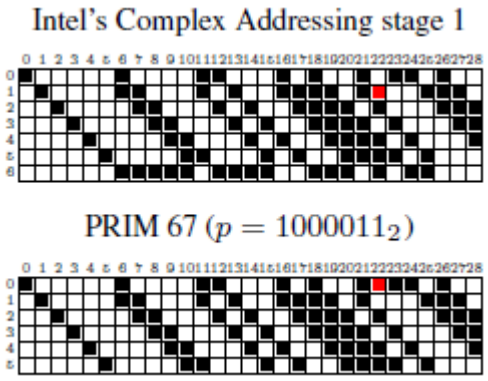


Figure 8.16. *Intel Complex Addressing stage 1 and PRIM 67*

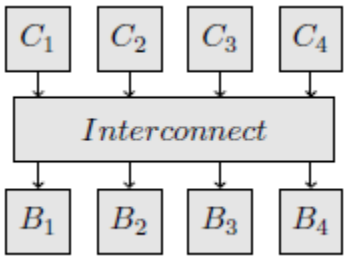


Figure 8.17. *Overview of the Kalray MPPA simplified local memory architecture*

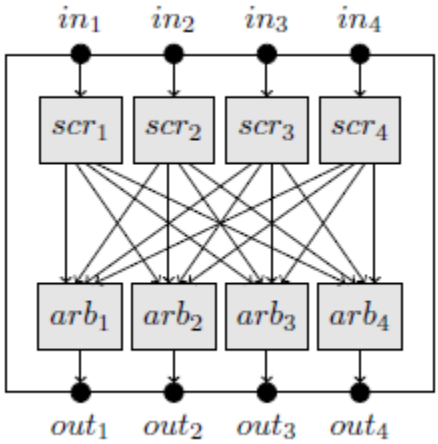


Figure 8.18. *Kalray MPPA simplified crossbar internal architecture*

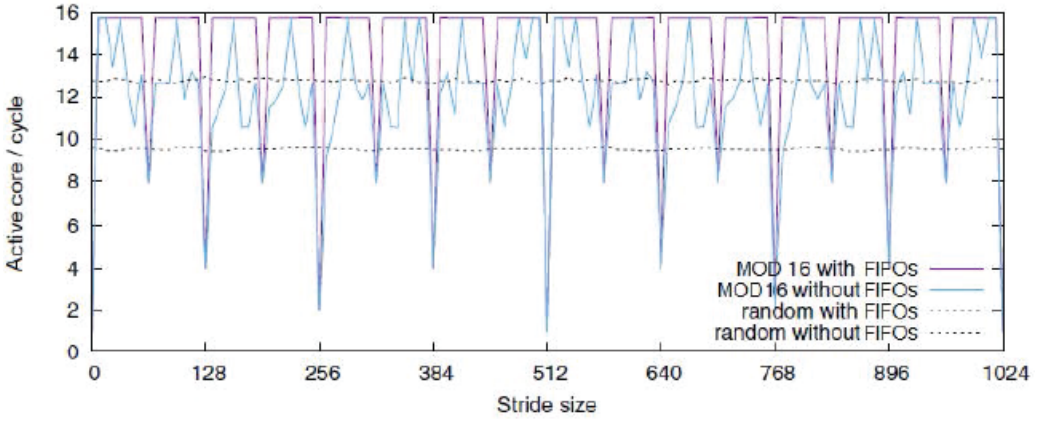


Figure 8.19. *Theoretical performance measure (in accesses per cycle) for stride access with MOD 16*

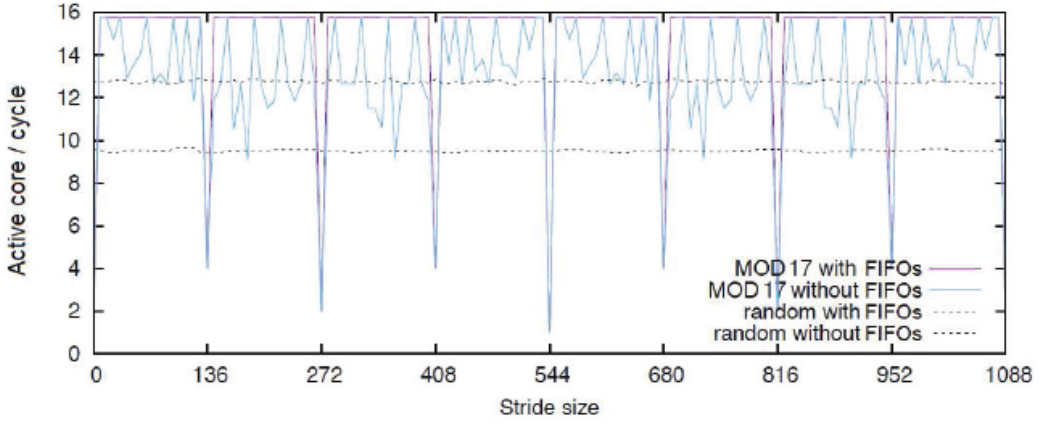
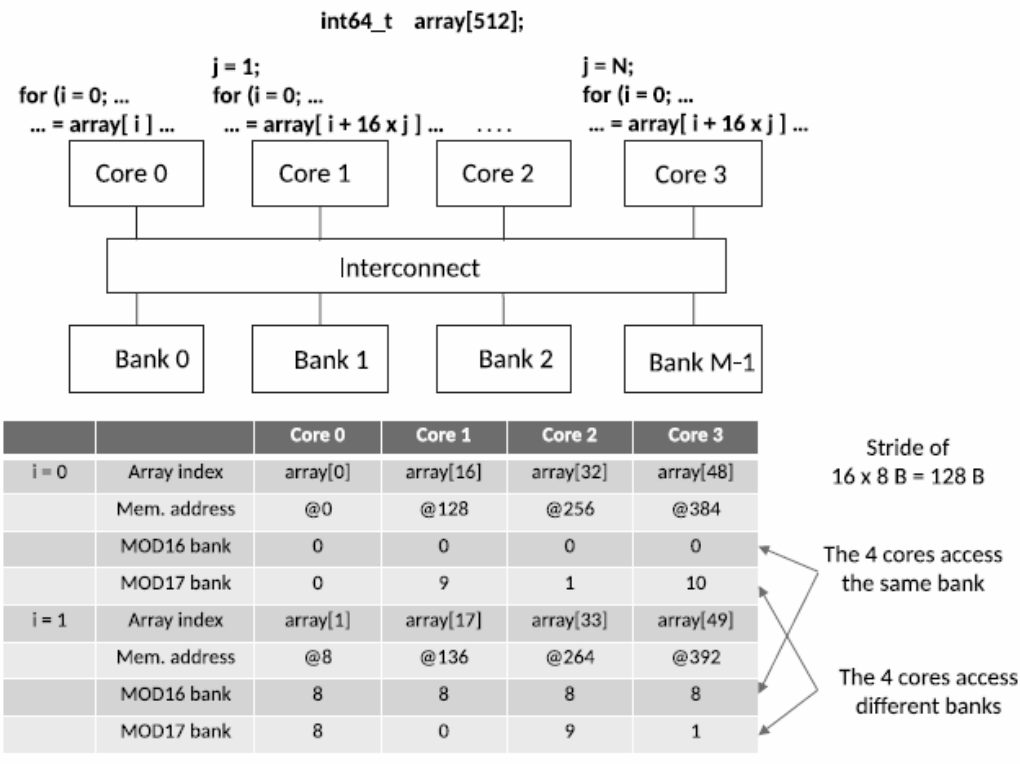


Figure 8.20. *Theoretical performance measure (in accesses per cycle) for stride access with MOD 17*



		Core 0	Core 1	Core 2	Core 3
i = 0	Array index	array[0]	array[16]	array[32]	array[48]
	Mem. address	@0	@128	@256	@384
	MOD16 bank	0	0	0	0
	MOD17 bank	0	9	1	10
i = 1	Array index	array[1]	array[17]	array[33]	array[49]
	Mem. address	@8	@136	@264	@392
	MOD16 bank	8	8	8	8
	MOD17 bank	8	0	9	1

Stride of
16 x 8 B = 128 B

The 4 cores access the same bank

The 4 cores access different banks

Figure 8.21. *Comparison between MOD 16 and MOD 17 for the same executable code and the same architecture*

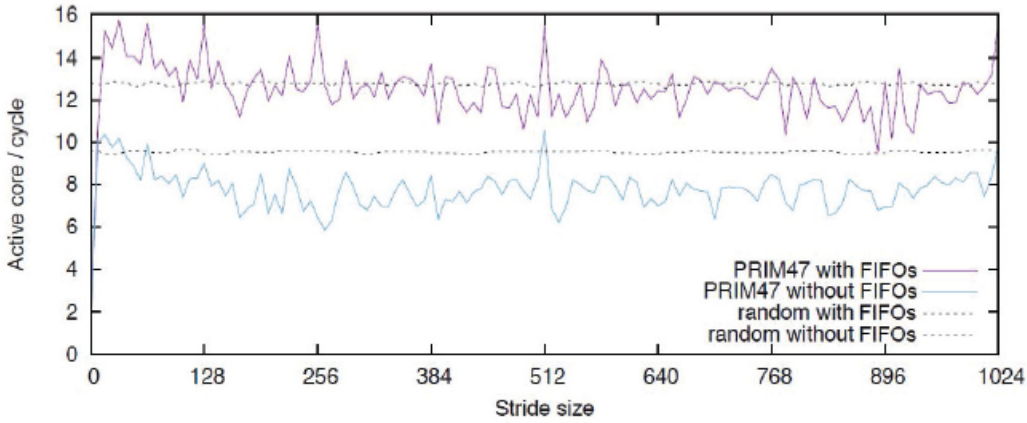


Figure 8.22. *Theoretical performance measure (in accesses per cycle) for stride access with PRIM 47*

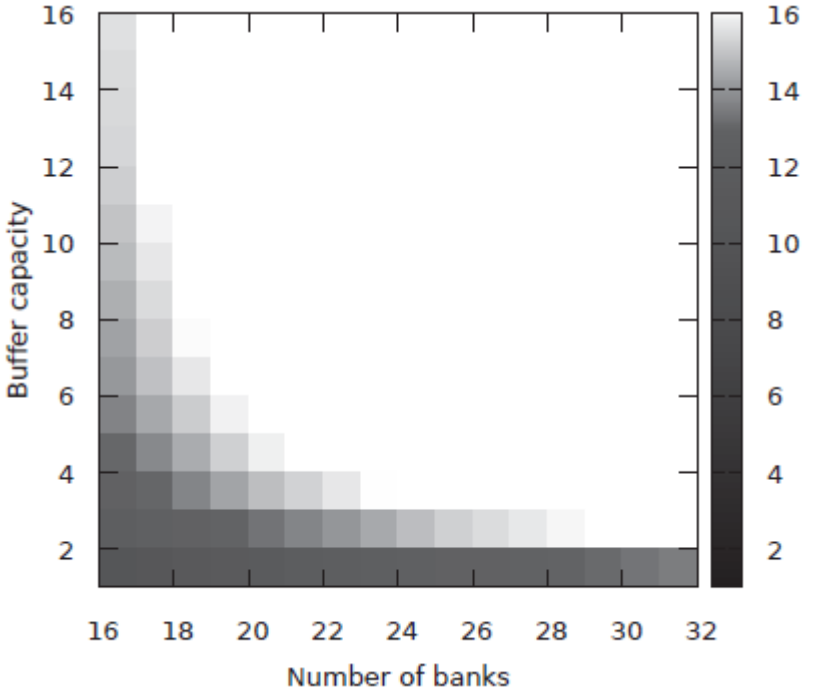


Figure 8.23. *Hotmap of memory access efficiency according to the number of banks and buffer size during random access patterns*

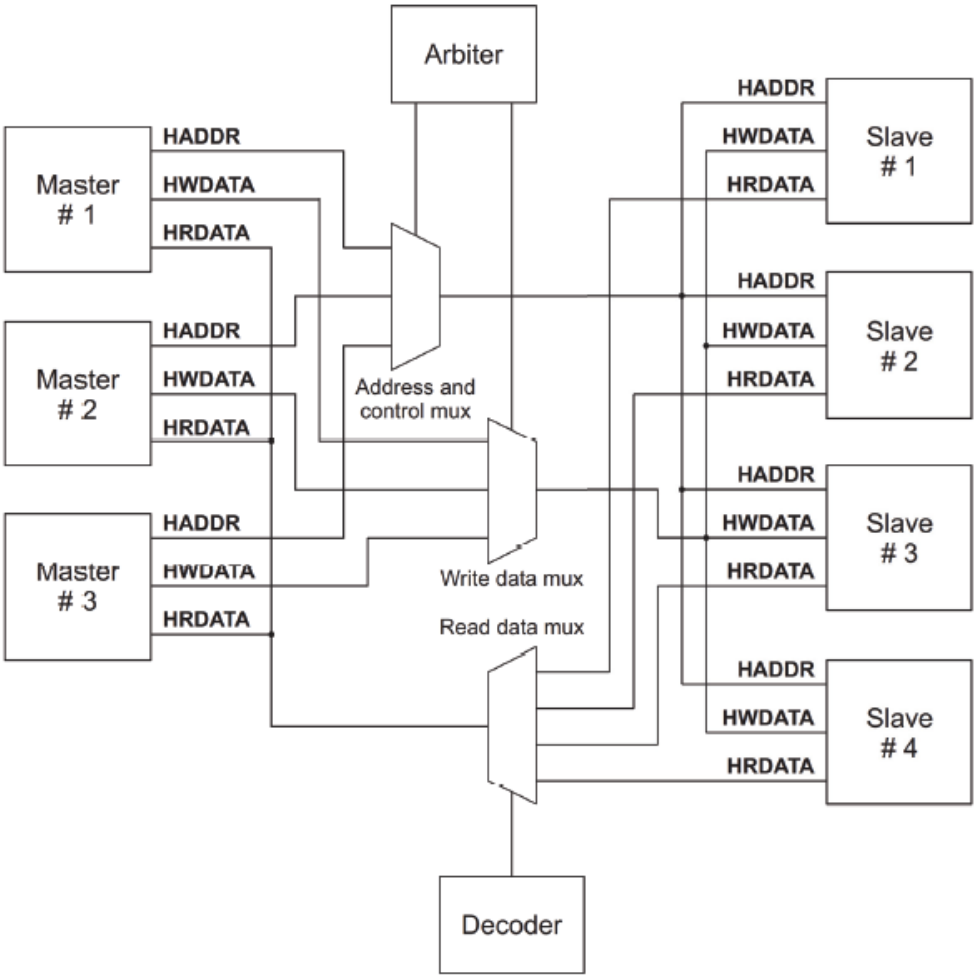


Figure 9.1. *A traditional synchronous bus, in this case the implementation of an AMBA AHB bus, requires centralized arbiter and decoder logic (ARM 1999)*

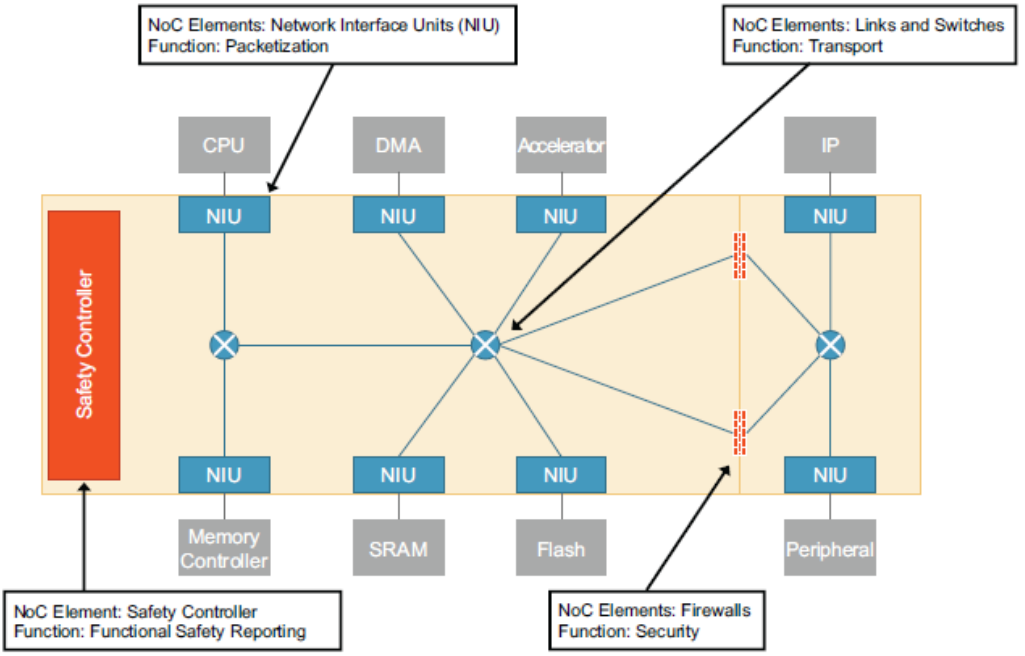


Figure 9.2. *Arteris switch fabric network (Arteris IP 2020)*

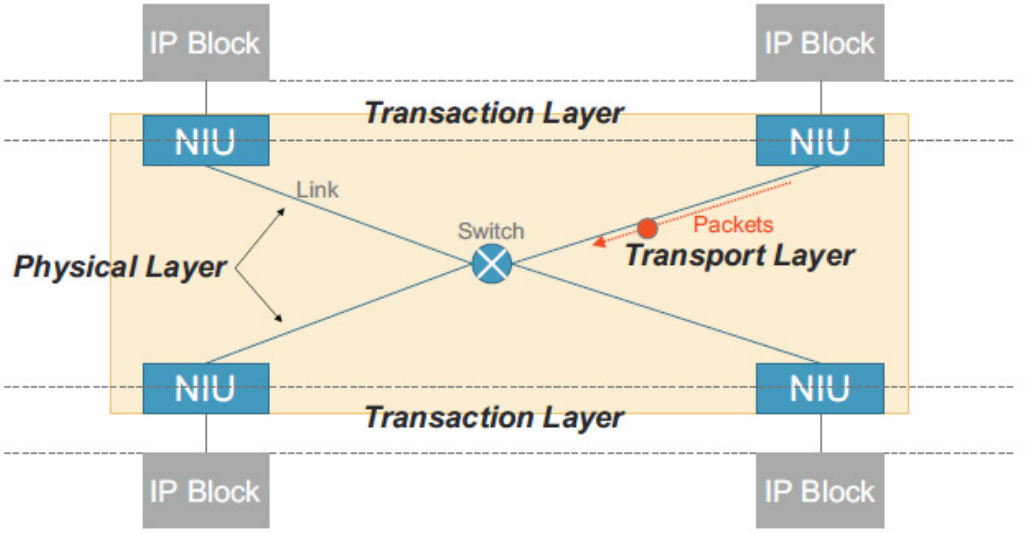


Figure 9.3. *NoC layer mapping summary (Arteris IP 2020)*

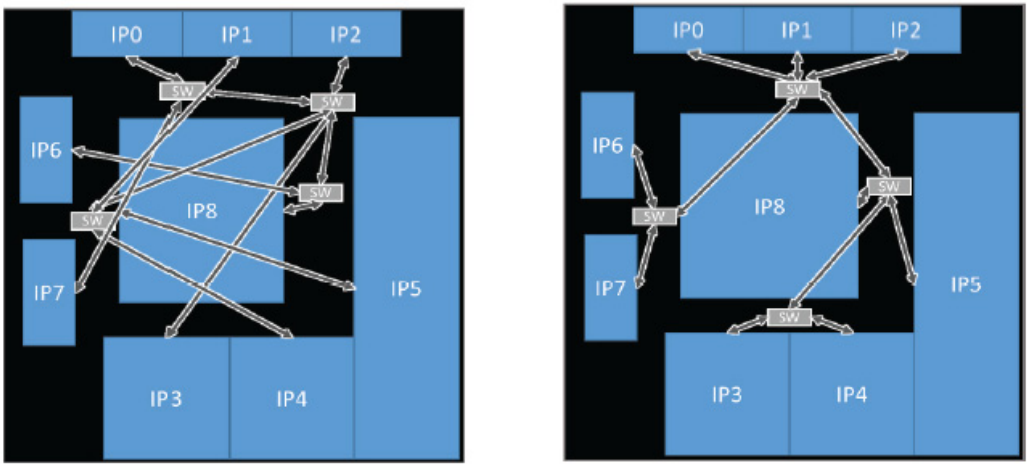


Figure 9.4. *The NoC on the left has a floorplan-unfriendly topology, whereas the NoC on the right has a topology that is floorplan-friendly (Arteris IP 2020)*

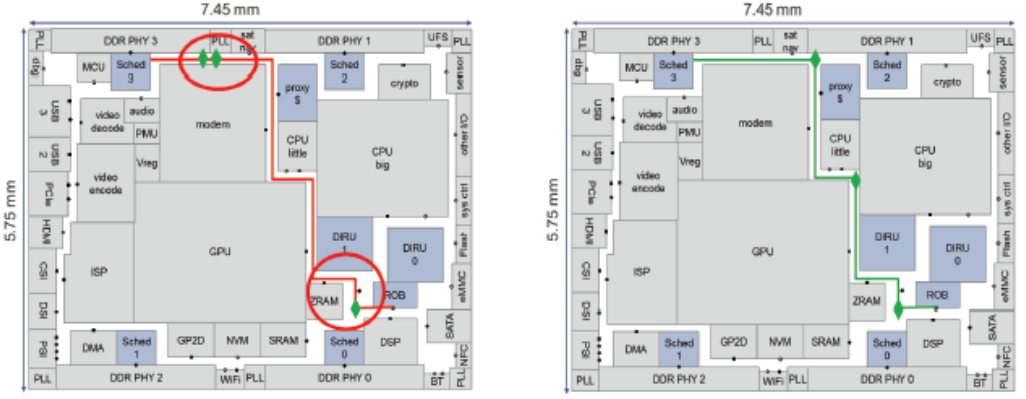


Figure 9.5. *Pipeline stages are required in a path to span a particular distance given the clock period and transport delay (Arteris IP 2020)*

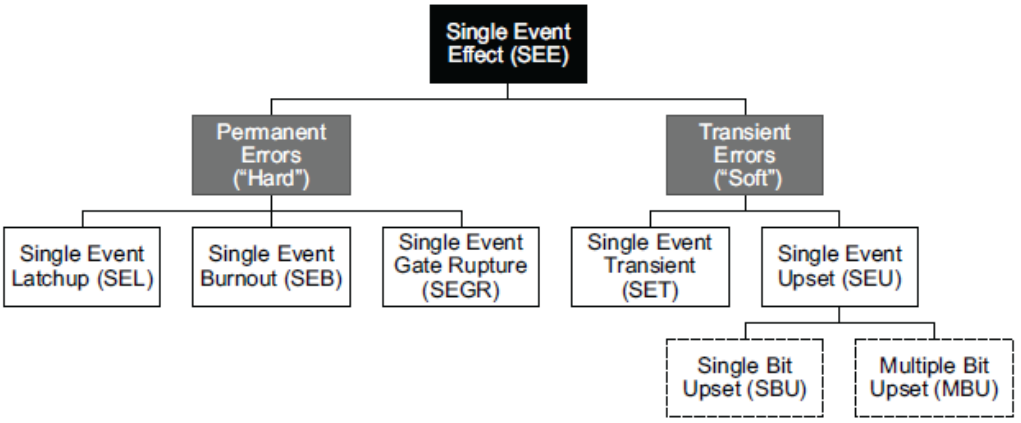


Figure 9.6. *Single-event effect (SEE) error hierarchy diagram (ISO26262-11 2018b; JESD86A 2006)*

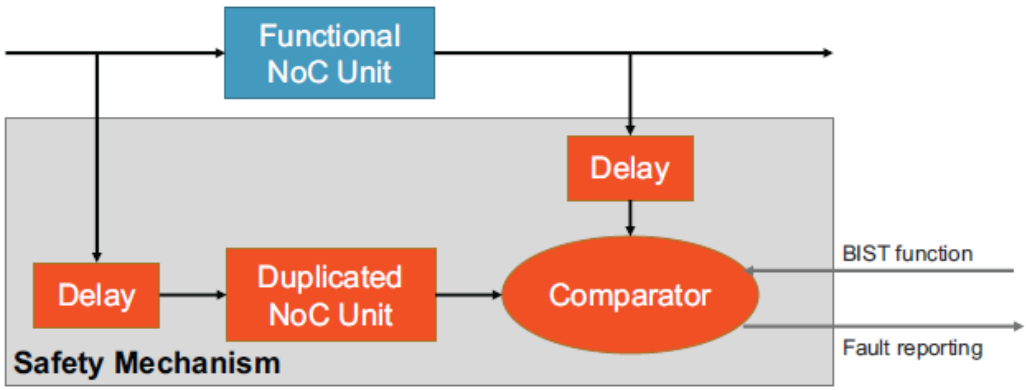


Figure 9.7. *Failure mode effects and diagnostic analysis (FMEDA) includes analysis of safety mechanisms, such as hardware unit duplication and a built-in self-test (BIST) (Arteris IP 2020)*

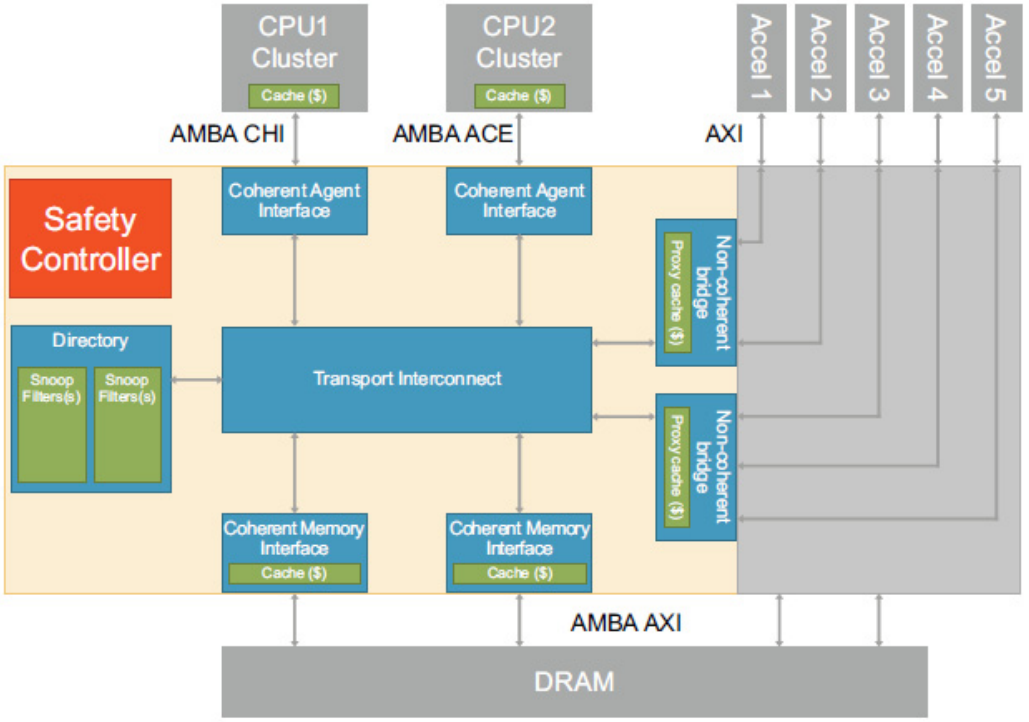


Figure 9.8. *A cache coherent NoC interconnect allows the integration of IP using multiple protocols (Arteris IP 2020)*

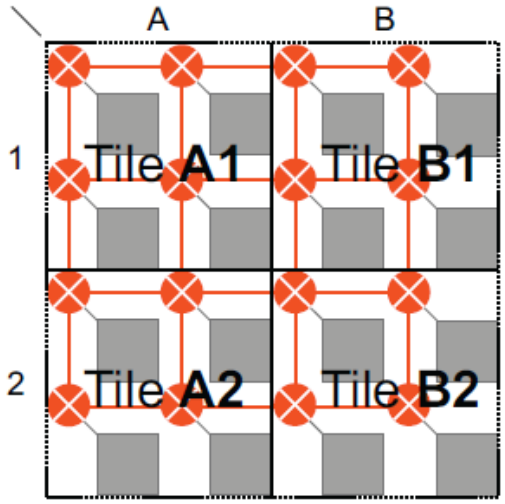


Figure 9.9. *NoC interconnects enable easier creation of hard macro tiles that can be connected at the top level for scalability and flexibility (Arteris IP 2020)*

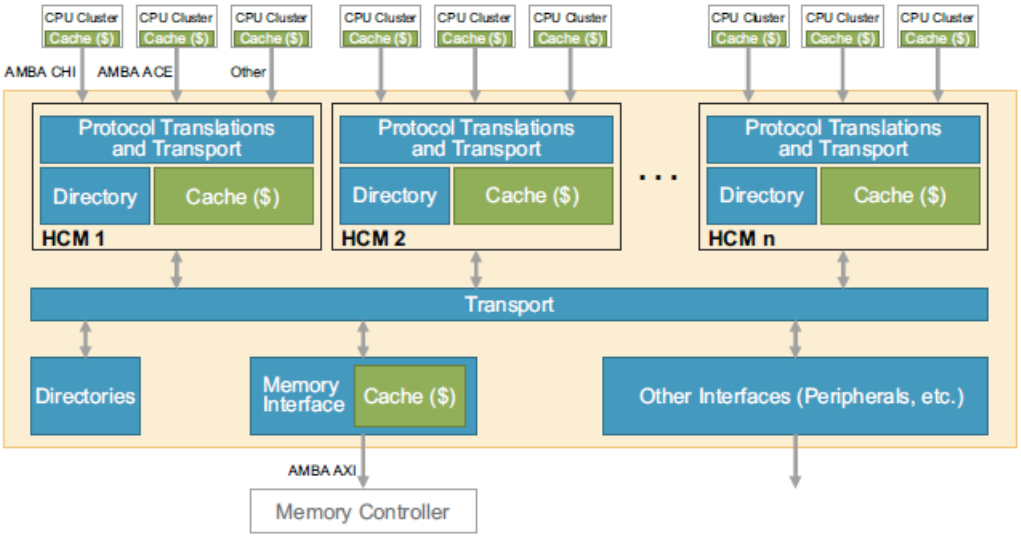


Figure 9.10. *Hierarchical coherency macros enable massive scalability of cache coherent system (Arteris IP 2020)*

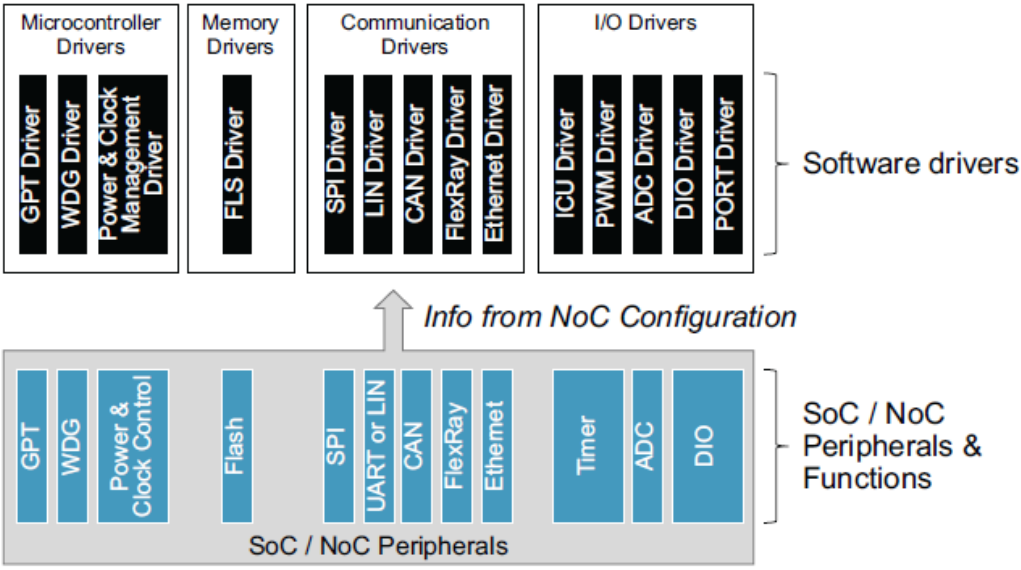


Figure 9.11. *An AUTOSAR MCAL showing how NoC configuration information will be used as a "golden source" for software driver header files. Adapted from Renesas (2020)*

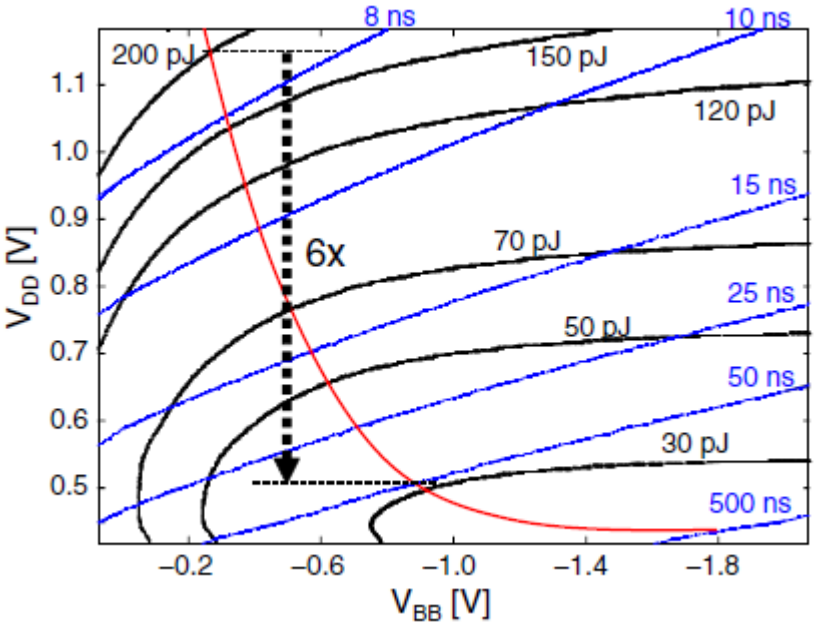


Figure 10.1. *Energy efficiency improvement by near-threshold computing*

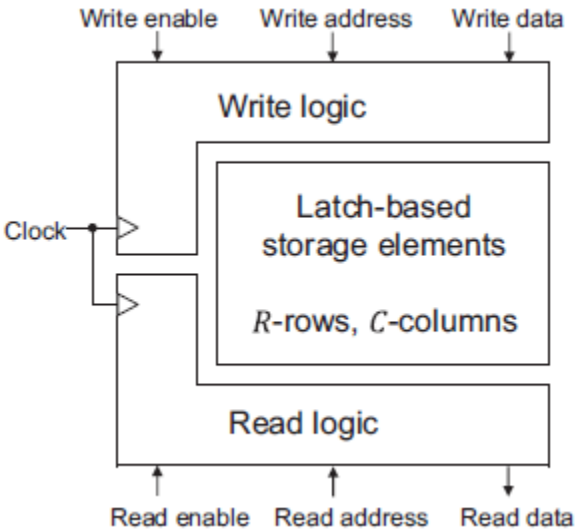


Figure 10.2. *Block diagram of the SCM with an* R × C*-bit capacity*

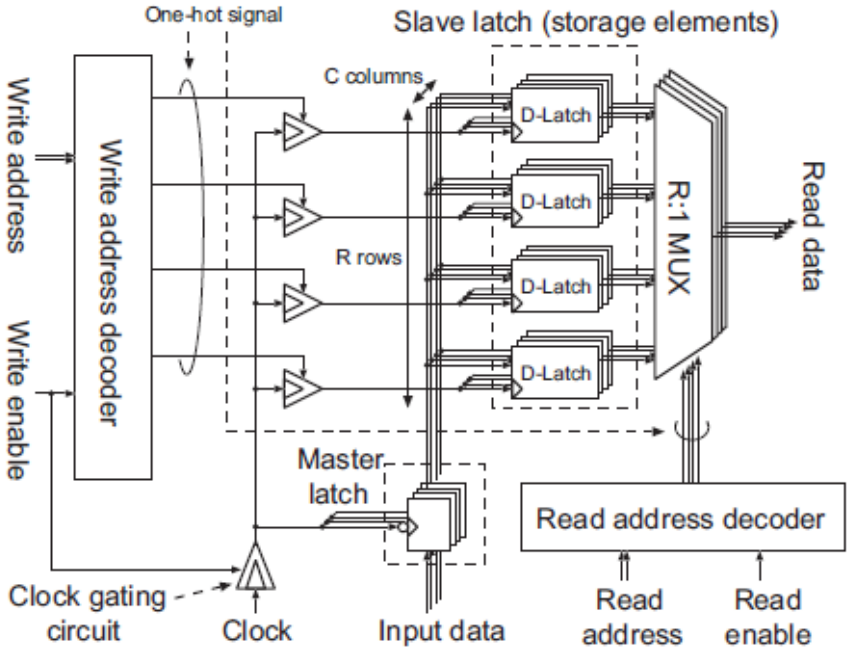


Figure 10.3. *An example of SCM structures (R = 4, C = 4)*

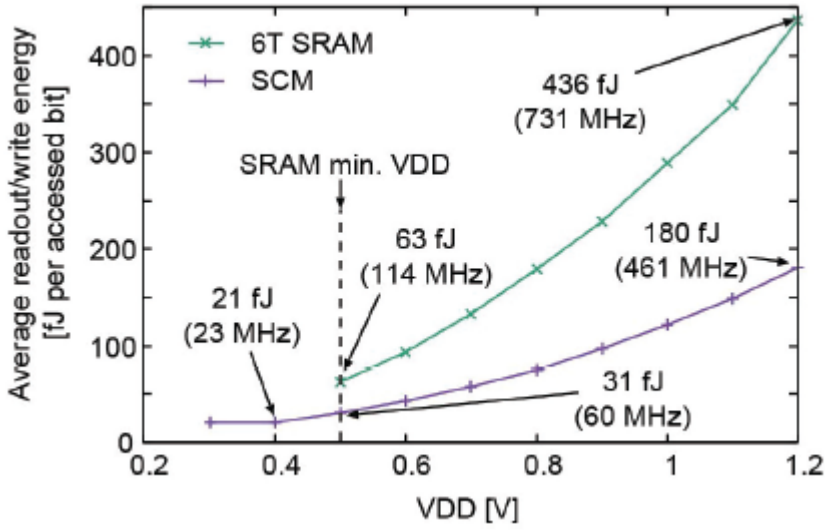


Figure 10.4. *Energy measurement results for two memories with a 256 × 32 capacity in a 65 nm low-Vt SOI-CMOS technology. 6T SRAM: 6-transistor SRAM. SCM: standard-cell-based memory*

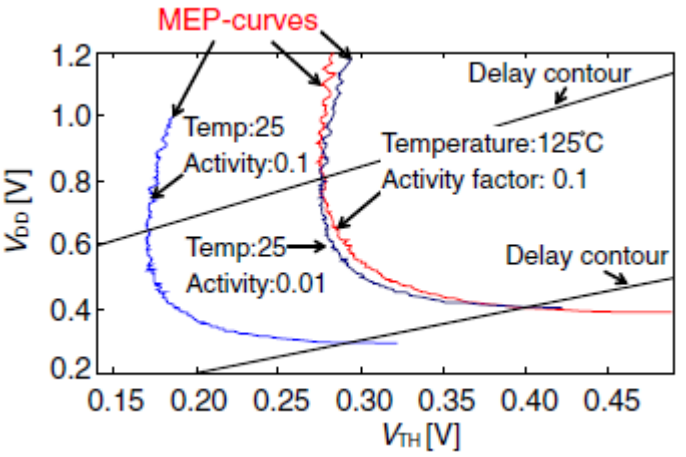


Figure 10.5. *Minimum energy point curves*

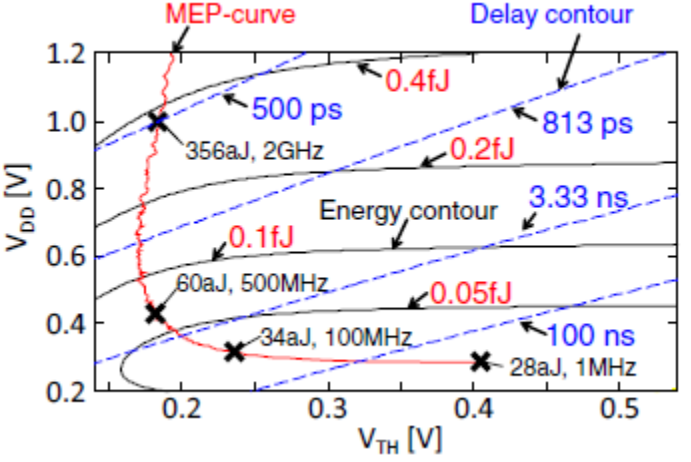


Figure 10.6. *Energy and delay contours*

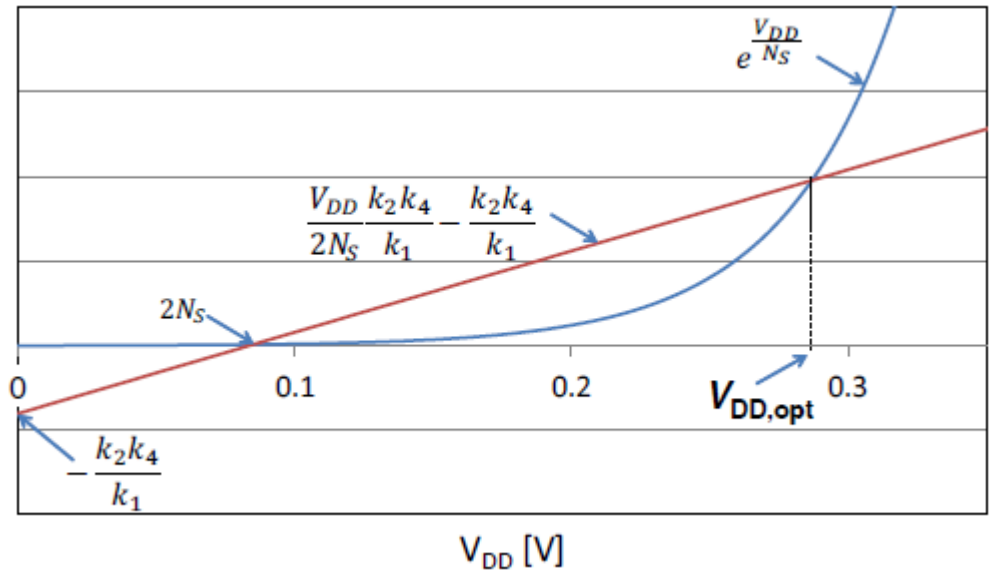


Figure 10.7. *Minimum energy point in near- or sub-threshold region*

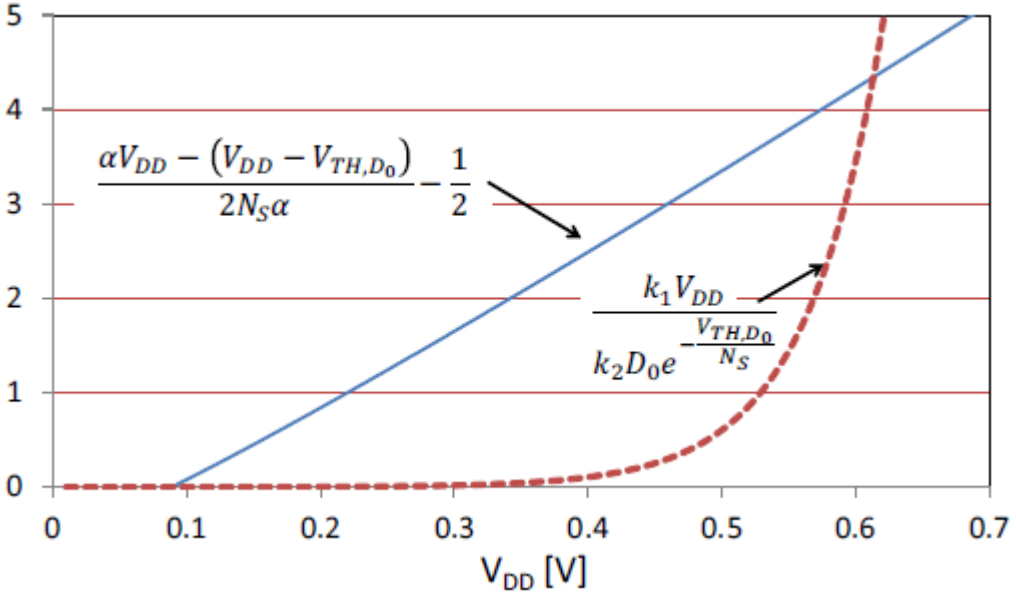


Figure 10.8. *Minimum energy point in super-threshold region*

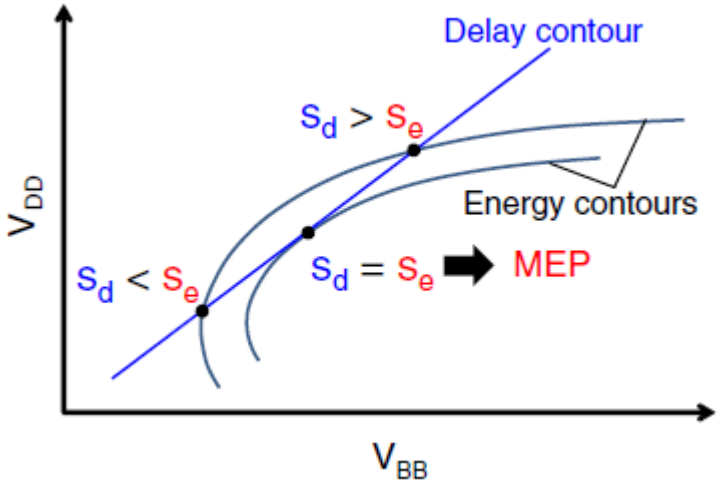


Figure 10.9. *The concept of minimum energy point tracking algorithm*

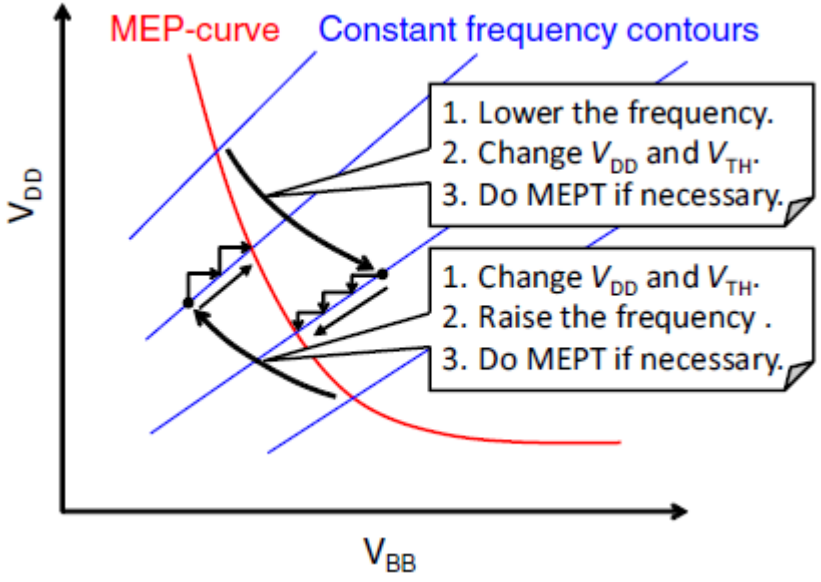


Figure 10.10. *An OS-based algorithm for MEPT*

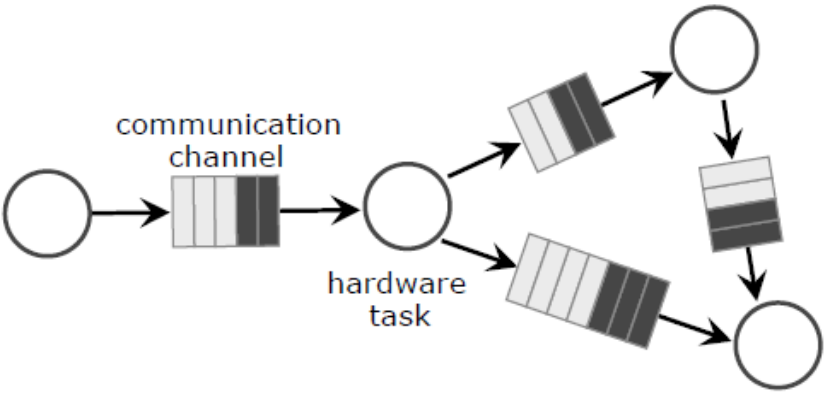


Figure 11.1. *Tasks in a heterogeneous computing architecture communicate with each other via FIFO-based channels*

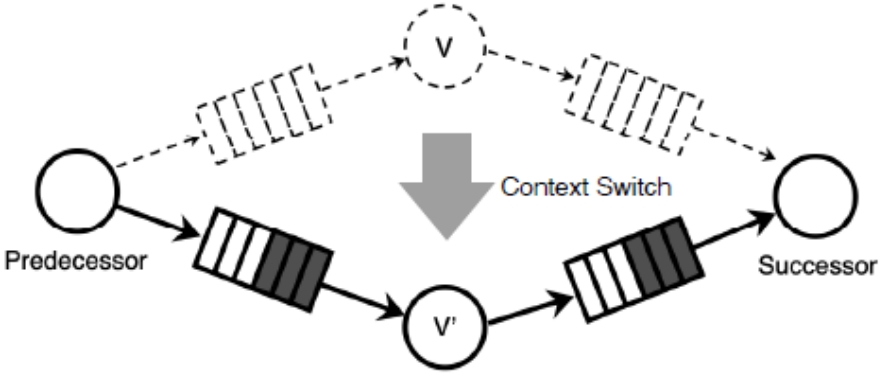


Figure 11.2. *Hardware context switch on a task with FIFO-based communication channels*

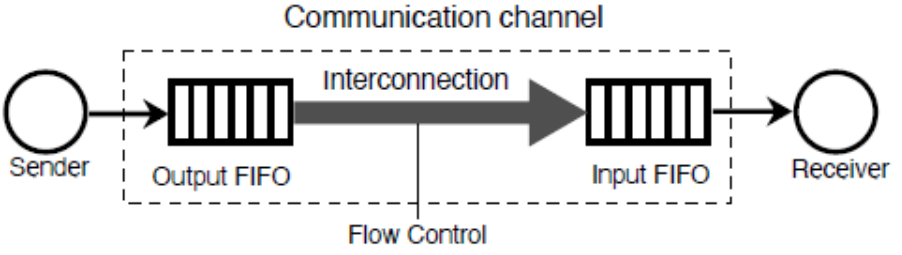


Figure 11.3. *Modified communication channel to support hardware context switching*

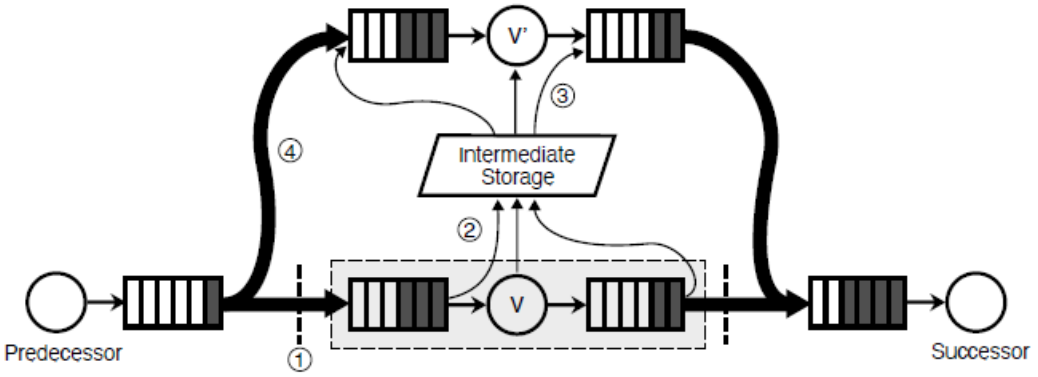


Figure 11.4. *The proposed communication protocol in hardware context switching*

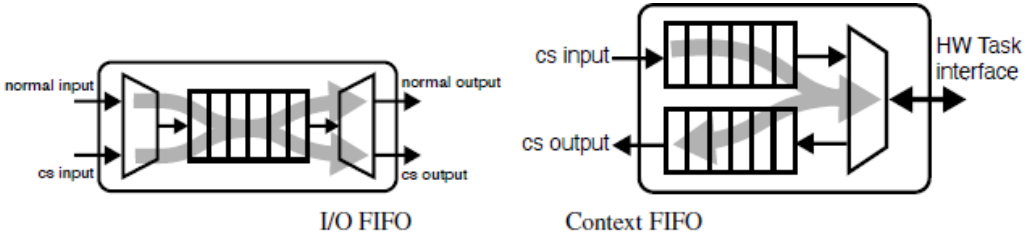


Figure 11.5. *FIFOs in the communication channel*

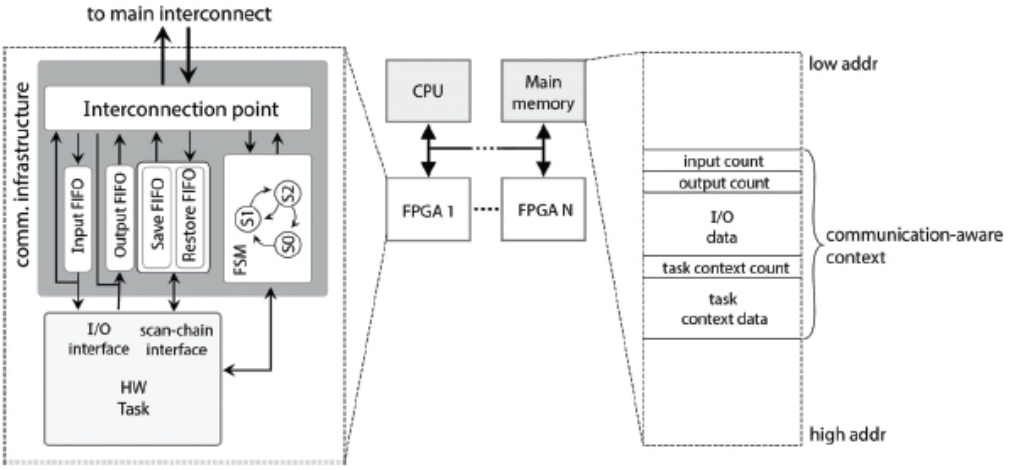


Figure 11.6. *Reconfigurable architecture with the proposed communication structure*

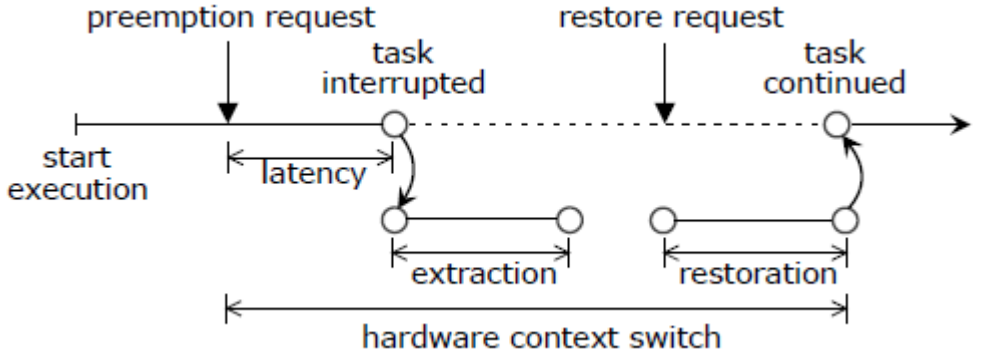


Figure 11.7. *Hardware context switch scenario in the experiments*

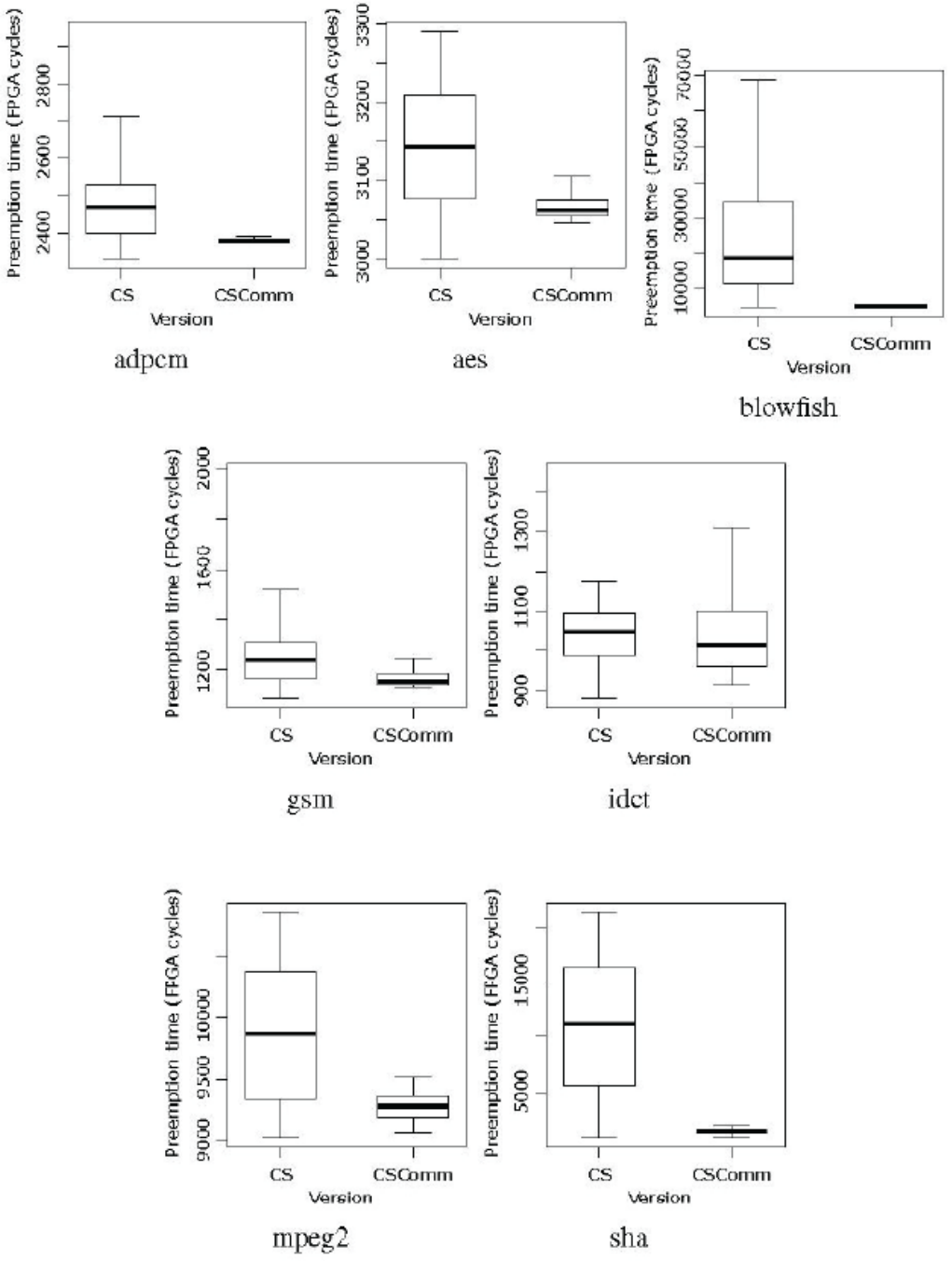


Figure 11.8. *Comparison of hardware context switch (preemption) time between CS and CSComm*

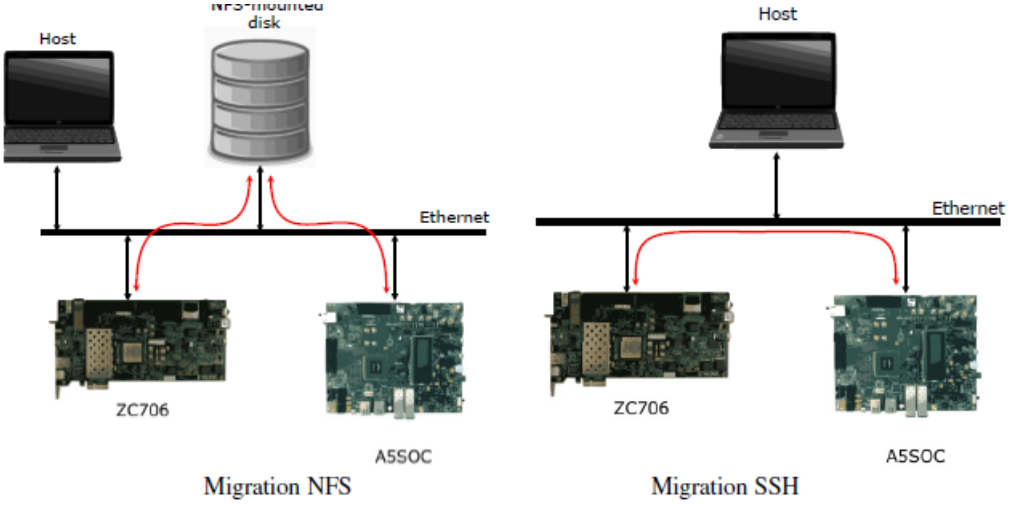


Figure 11.9. *Hardware task migration between heterogeneous reconfigurable SoCs*

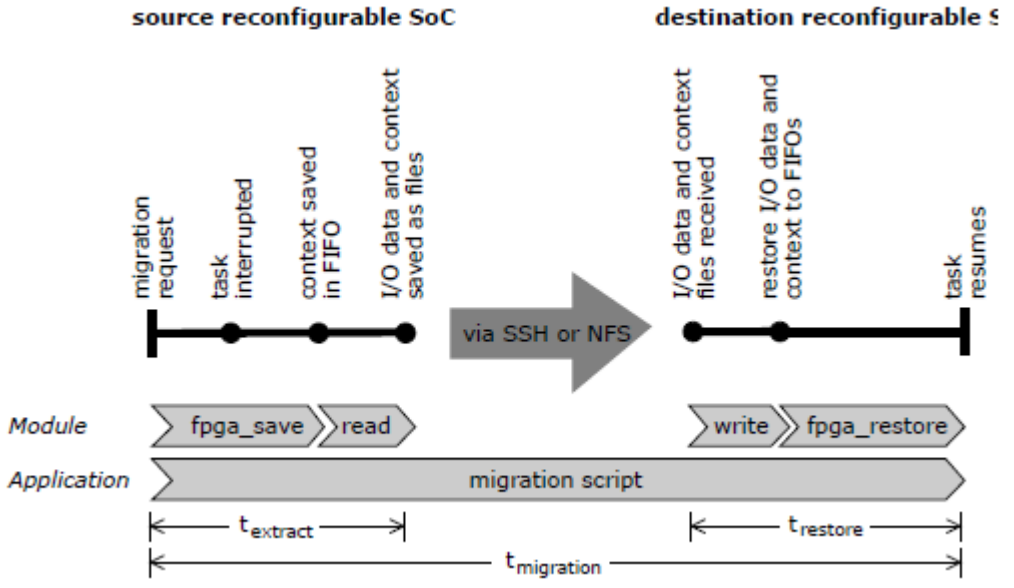


Figure 11.10. *Task migration timeline*