COMPILING HIGH THROUGHOUT NETWORK PROCESSORS
(PART OF THE GORILLA PROJECT)

Maysam Lavasani, Larry Dennison§, Derek Chiou
University of Texas at Austin
§ Lightwolf Technologies

GORILLA METHODOLOGY

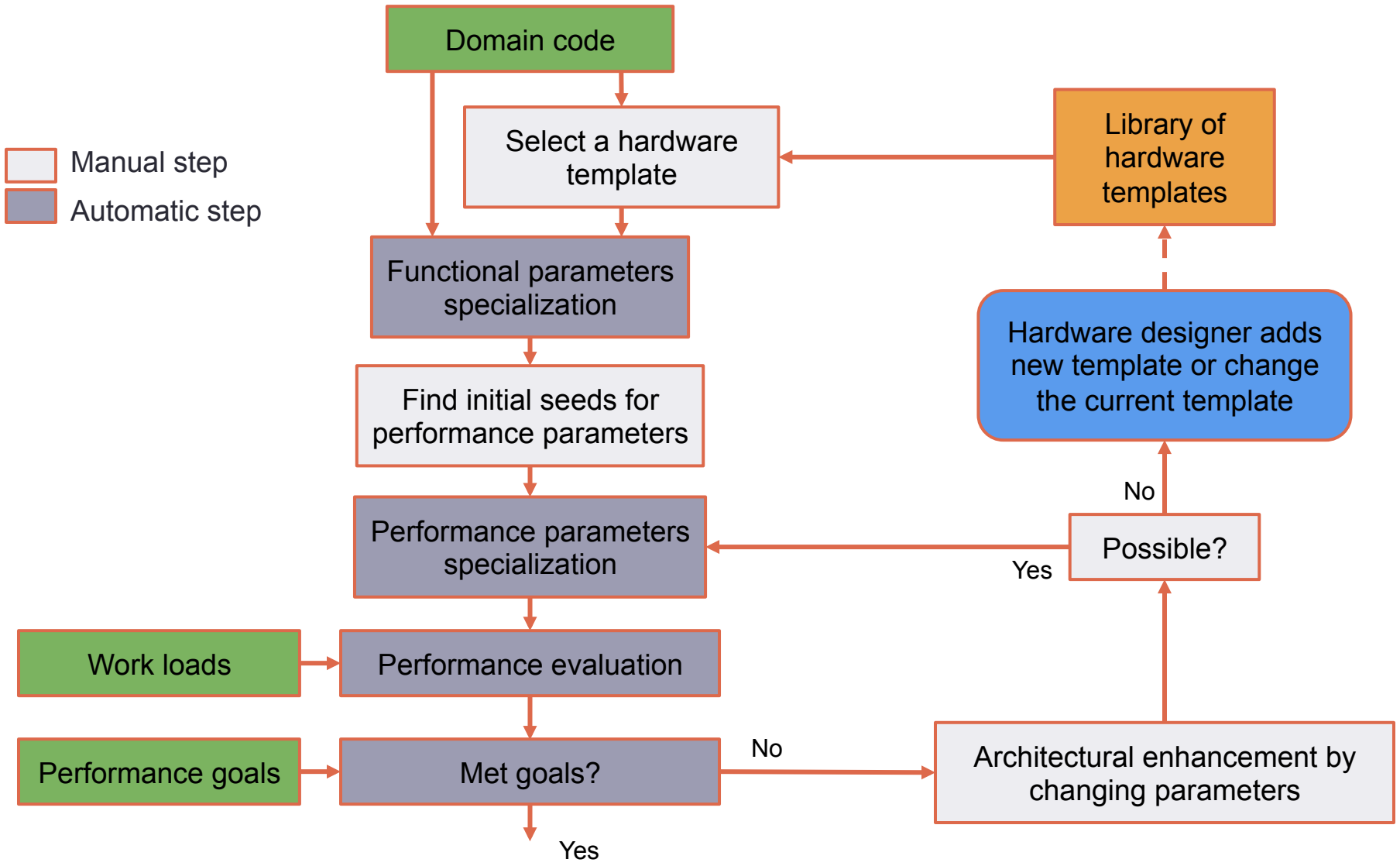
Gorilla in a Nutshell

- An FPGA solution for data parallel applications that are inefficient on CPUs and GPGPUs
  - Many data elements to process
  - Irregular data structures
    - E.g., trees, graphs
    - High throughput , random memory access requirements
  - Execution path highly data dependent
    - SIMD is inefficient (e.g. packet processing, iterative refinement)
  - Computation amenable to acceleration via specialized hardware
  - Original idea formed and patented in Avici Systems (patent# 7823091)

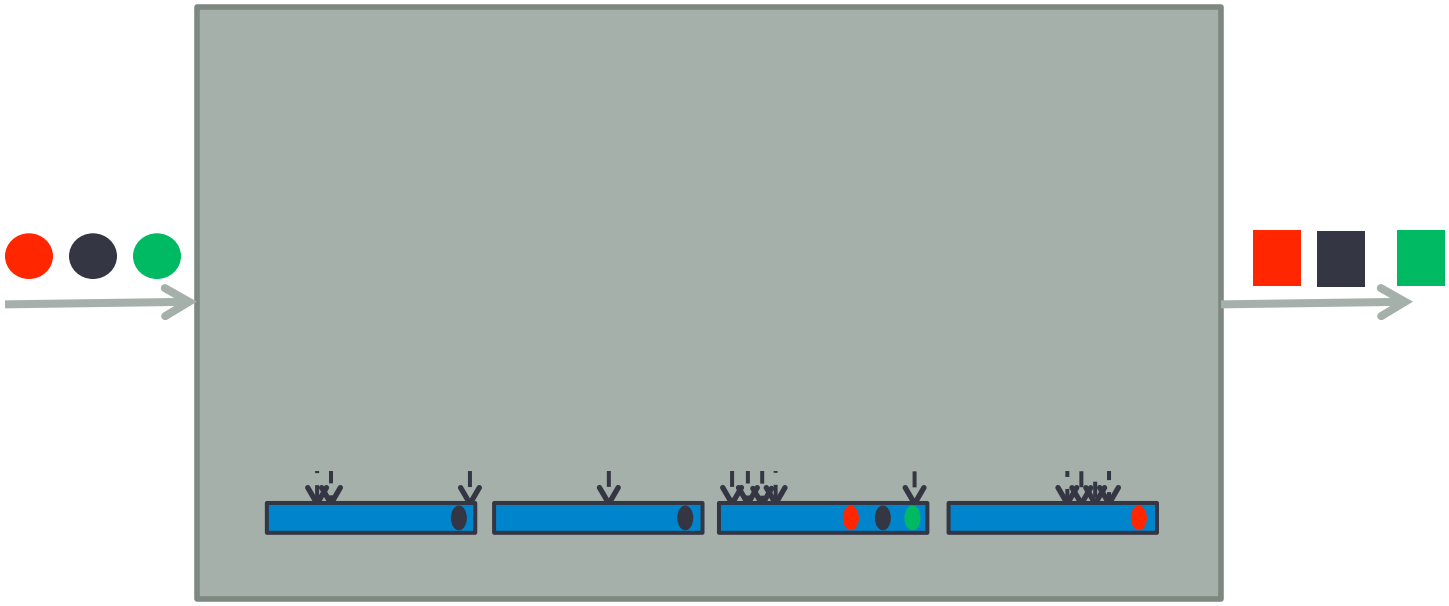
Gorilla Methodology

- Domain expert writes in stylized C (domain code)
- Hardware designer writes parameterized hardware components (hardware template)
  - Parameters may be values or code
  - Functional parameters
    - E.g., IPv4 code or IPv6 code
  - Performance parameters
    - Number of threads, packets in flight, scheduling policy
- Tools automatically
  - Merge domain code with hardware templates
  - Explore the design space to meet the design constraints
- Sharp contrast with C-to-gates approaches
  - Gorilla uses predesigned template for the target hardware

Gorilla Design Process

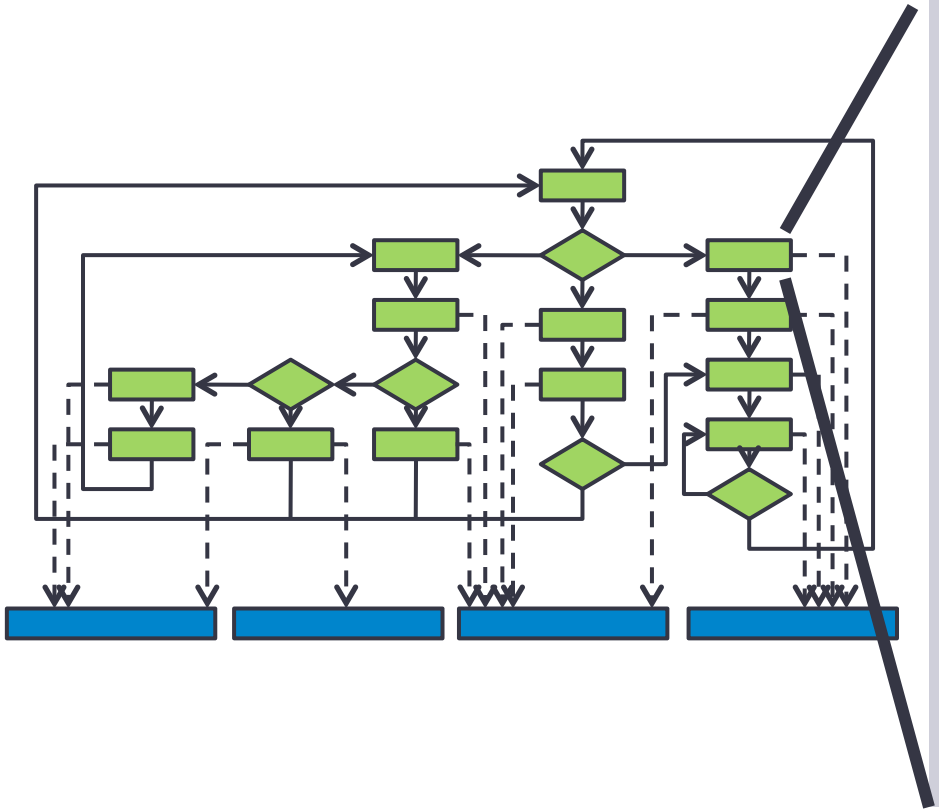


Execution Model



- A single processing kernel is applied on each input data element and generates an output data (infinite loop)
- kernel is modeled as a set of processing steps

Domain Code - Programming Model



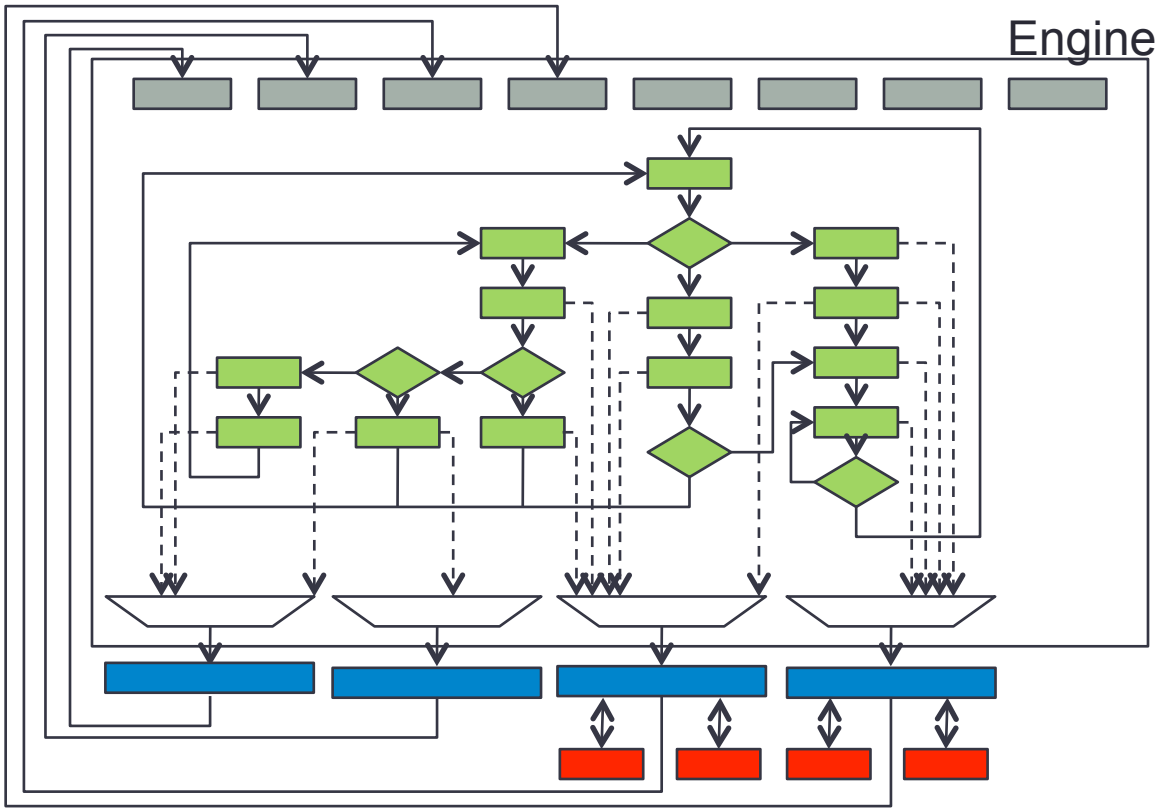
```
instr_addr_t  IP_CLASSIFY()
{   IP_protocol_t   wordx;

    //Lookup the destination address
    Dport = LOOKUPX.search(Da);

    //Read the TTL and Chksum fields
    wordx =  MEMX.read(PP, TTL_WORD);
    TTL = wordx.TTL;
    Chksum = wordx.CHKSUM;
    TTL = TTL – 1;
    Chksum = Update_chksum(Chksum);
    switch (TTL == 0) {
      case OK: Next_step = NP_INSTR_EMIT;
      default: Next_Step =
                    NP_INSTR_EXCEPTION;
}
}
```

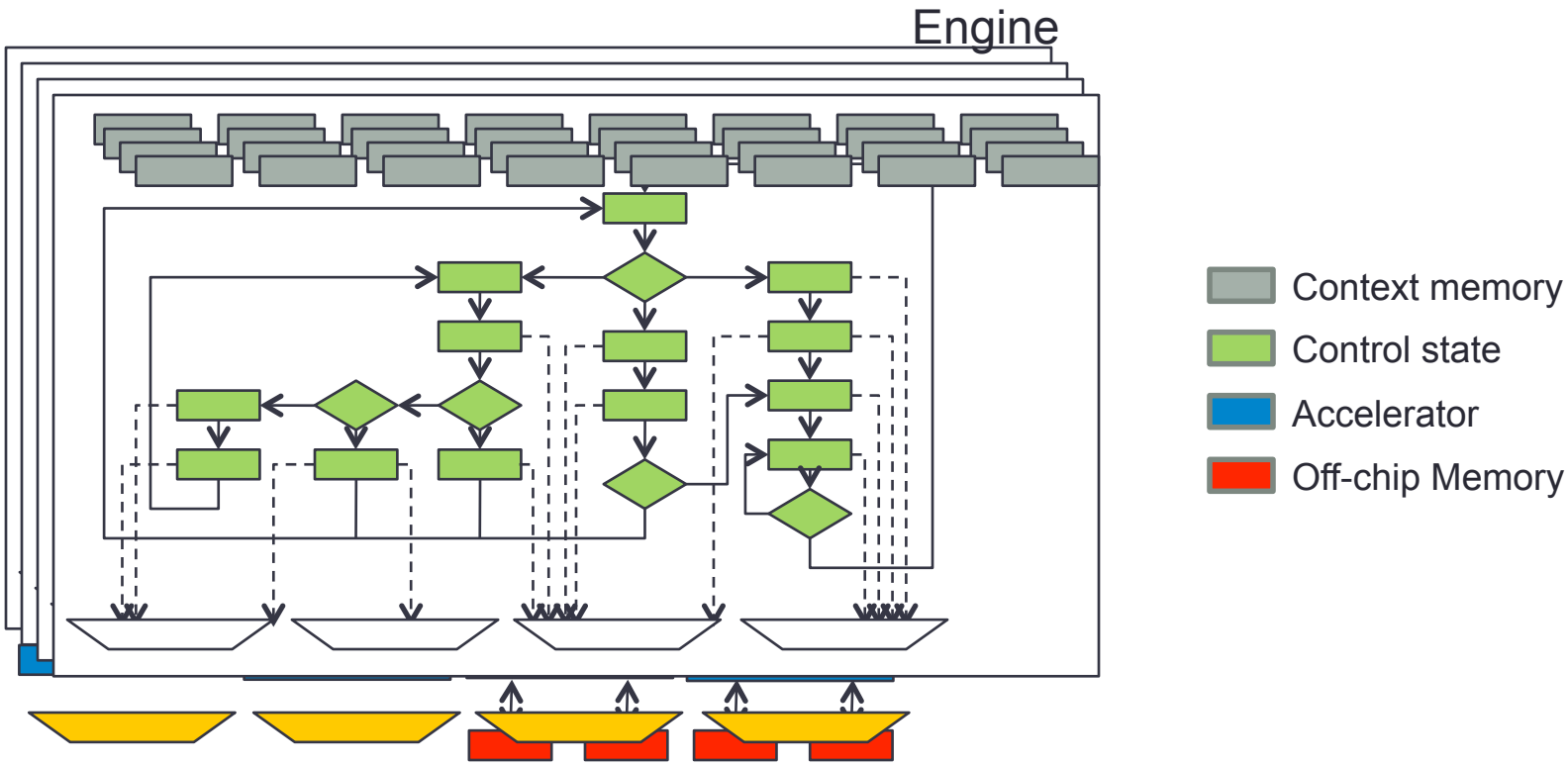
- Each step is written as a C function
  - Arithmetic/logic operation on program input and context variables (globally visible to all functions)
  - call to special purpose accelerators
  - Explicit, computed jump to next step

A Programmable Engine



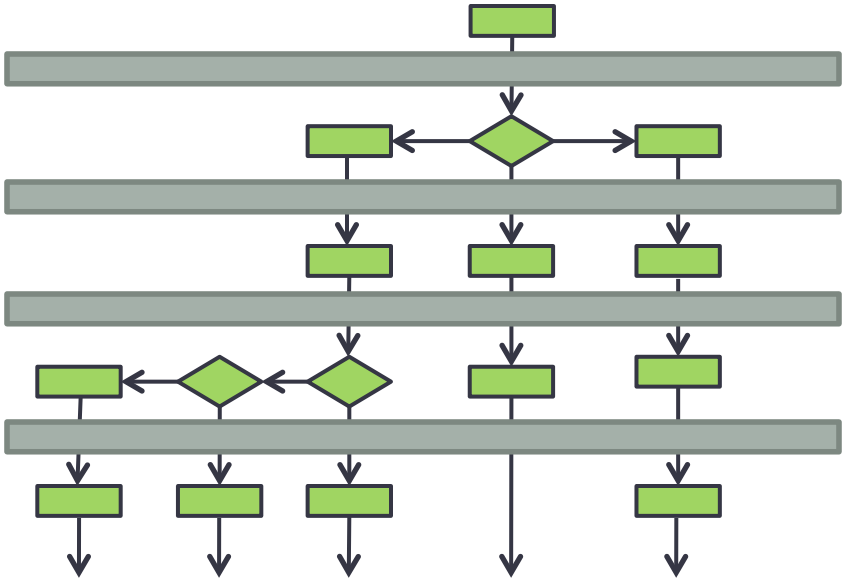
Context memory
Control state
Accelerator
Off-chip Memory

Scaling Throughput



- Multiple engine contexts
  - Multi-threaded engines
  - Multiple engines
- Contexts increase performance until an accelerator becomes the bottleneck

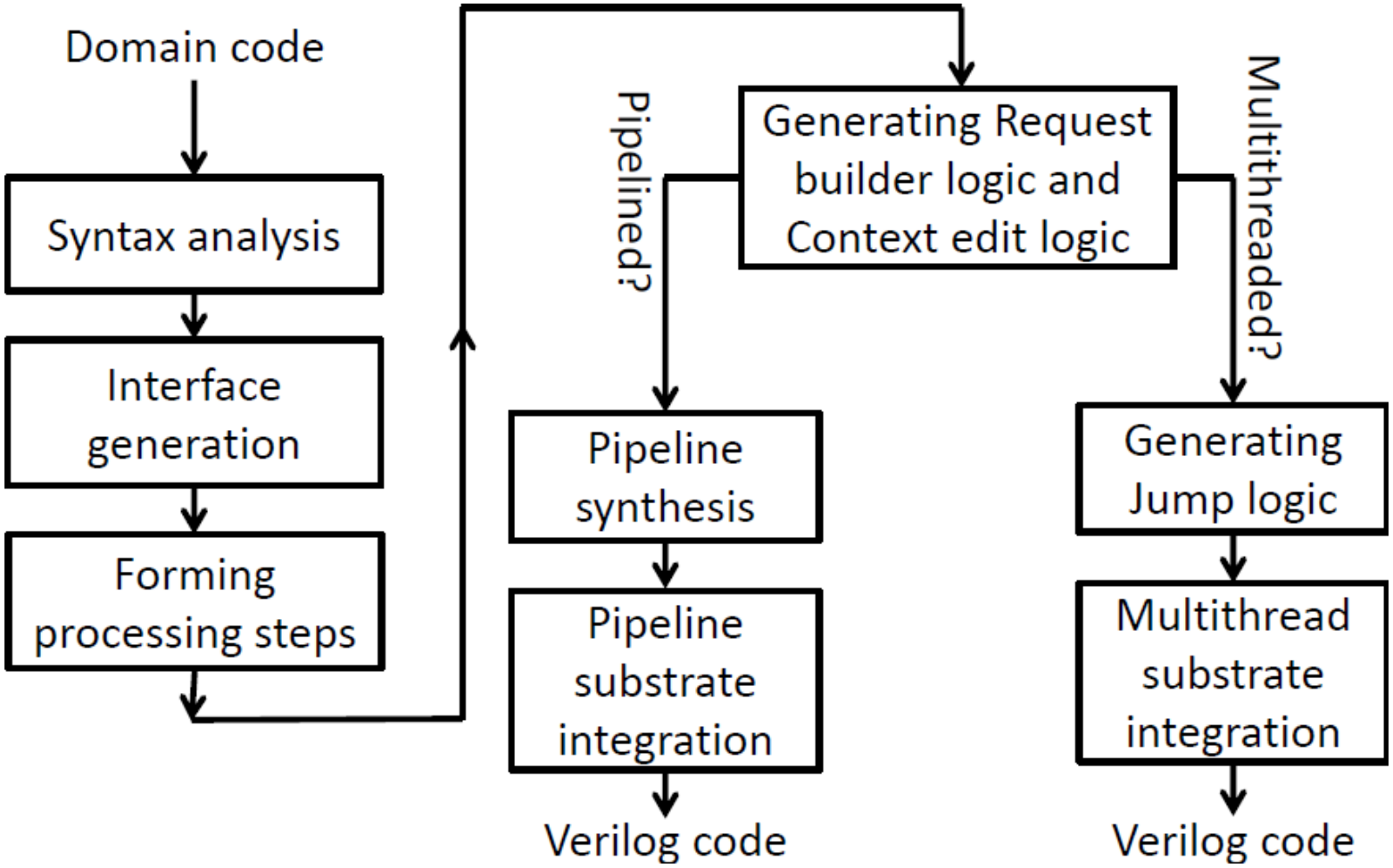
Pipelined Engines



- Multithreading the engines and engine duplication are not the only ways for increasing the throughput
- Pipelining the engine is possible providing
  - No backward jumps between processing steps
  - Access to an accelerator is restricted to a given pipe stage
- Pipelining has sometimes lower overhead than other solutions
  - No thread management overhead
  - No duplication overhead

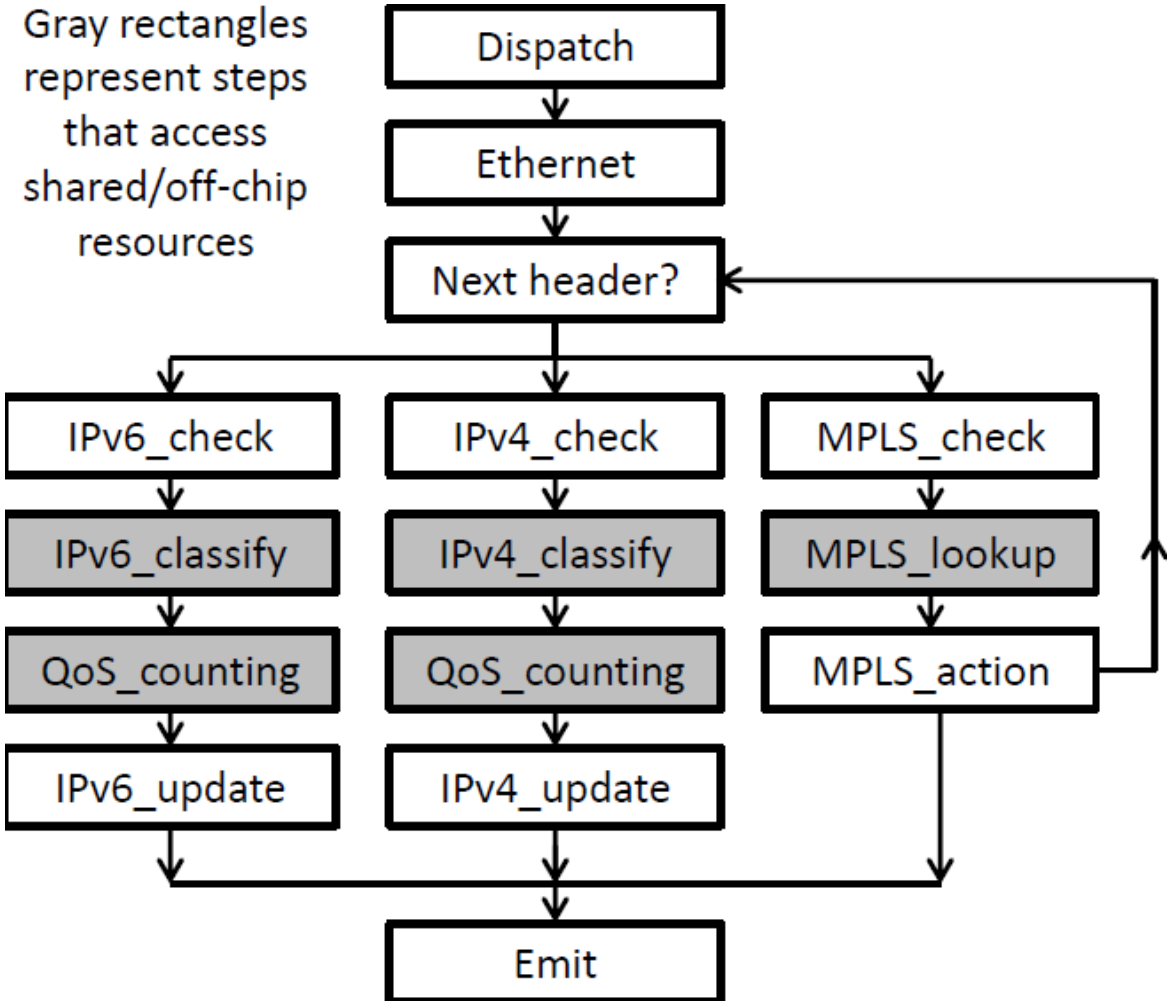
Gorilla Compilation Process

Written in ANTLR

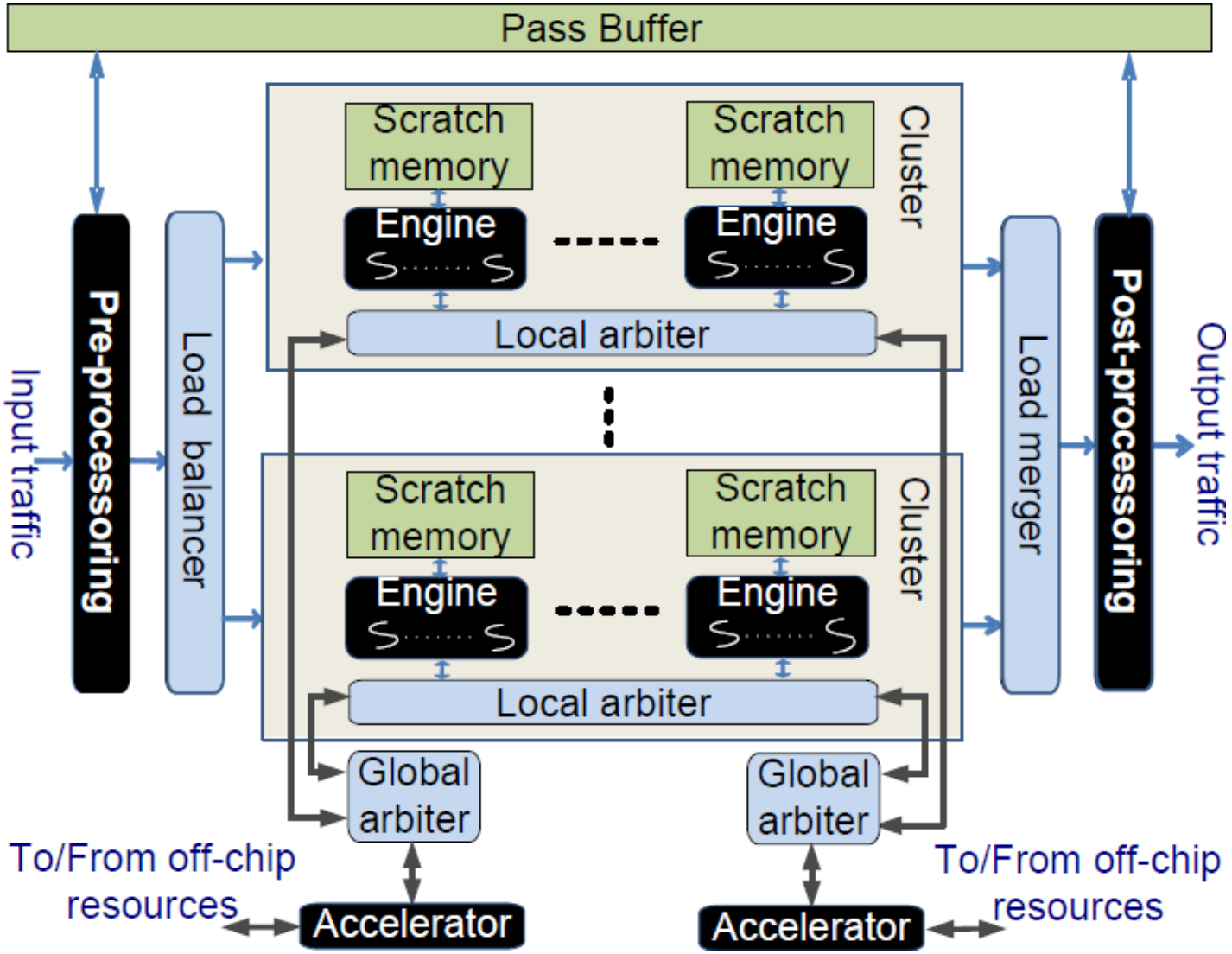


CASE STUDY: NETWORK PROCESSOR

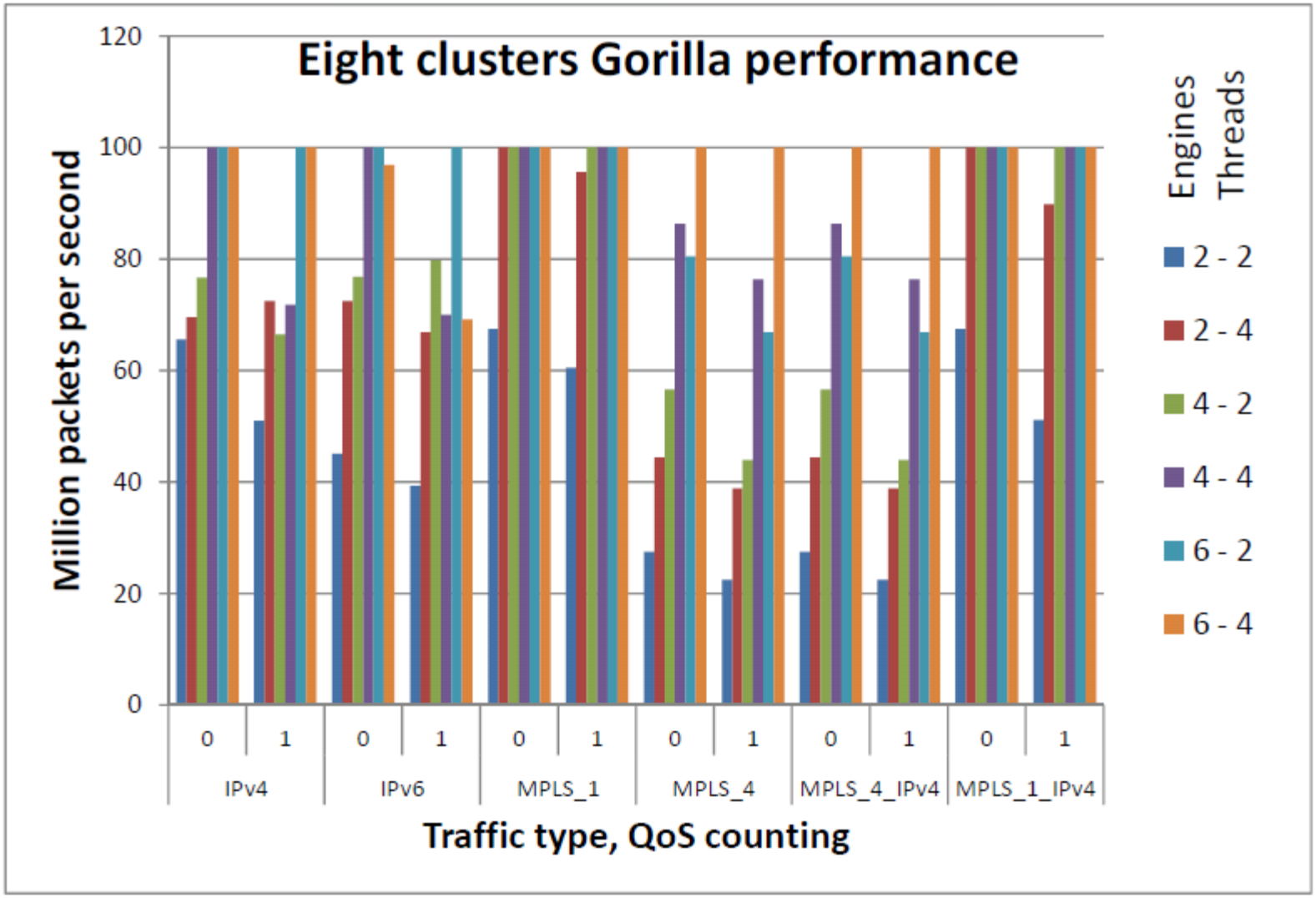
Packet Header Processing



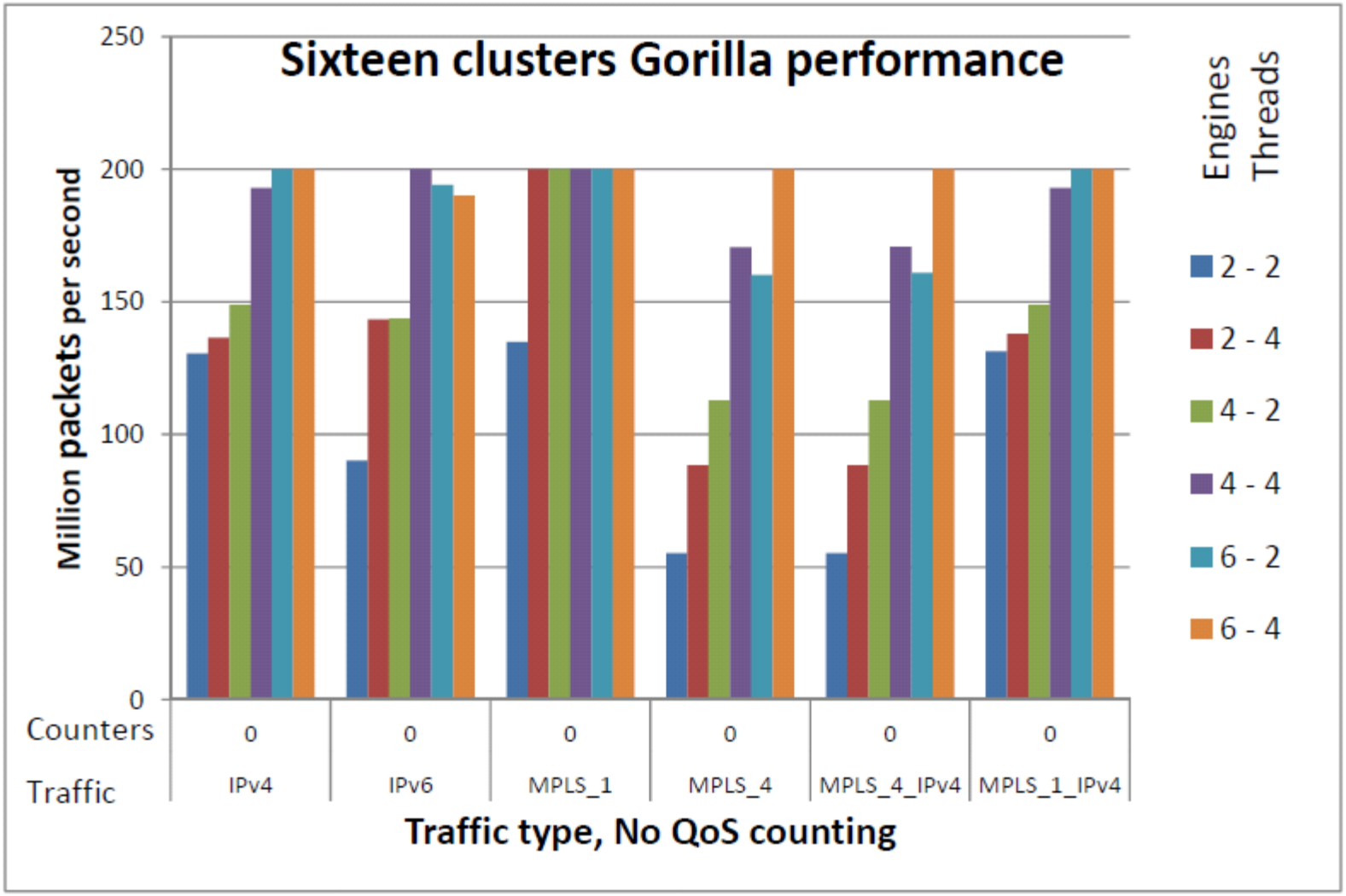
Packet Processing Hardware Template



Experimental Results - 100MPPS on Virtex-6 VHX380T



Experimental Results - 200MPPS on Virtex-7 VHX870T



Board Implementation

- Gorilla-generated network processor (IPv4 only) on ML605 (Xilinx Virtex-6 XC6VLX240T)
  - Emulate QDRs with accurate timing in BRAMs
- 16-3-2 configuration delivers 100MPPS
  - Most logic running at 100Mhz
  - Consistent with simulation results
- Random packet generator and statistics collector
- Measured core power (excluding the I/O) less than 4 watts

Comparing with CPU/GPGPU based systems

- Network processor on a single Xilinx Virtex-7 VHX870T FPGA
  - Achieved 200MPPS throughput (100 Gbps) for packet processing
  - More than six times the performance of 32 Nehalem cores
    - Routebricks[SOSP09]
  - Twice the performance of 8 Nehalem cores, 2 Nvidia GTX480 GPUs
    - Packetshader[Infocom10]

Conclusion

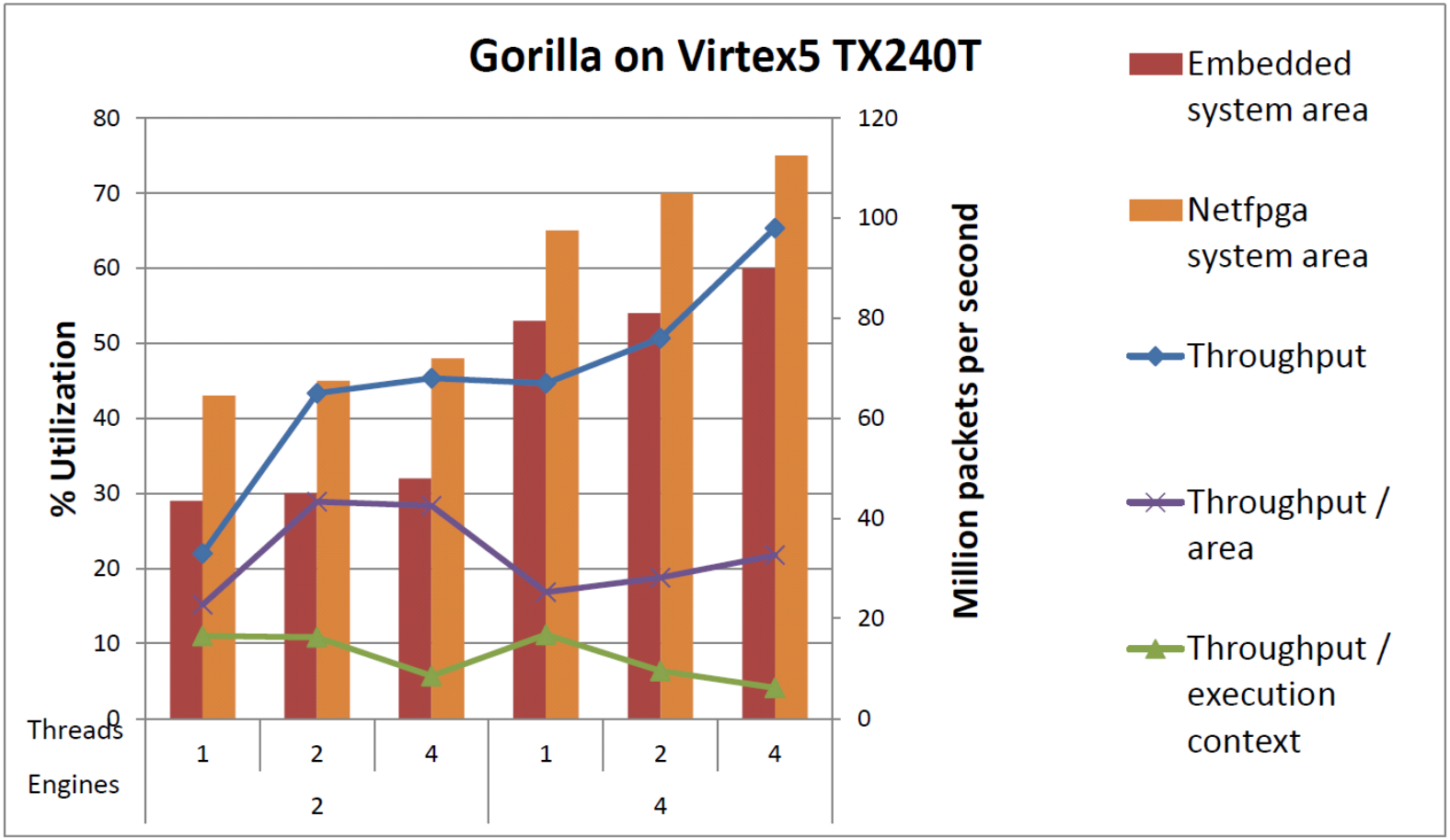
- A methodology for designing FPGA-based hardware for an interesting class of applications
- Network processing example is two orders of magnitude better power/performance than best CPU/GPGPU solutions
- Implementing additional applications using this infrastructure

Thank you

FPGA utilization IPv4/IPv6/MPLS

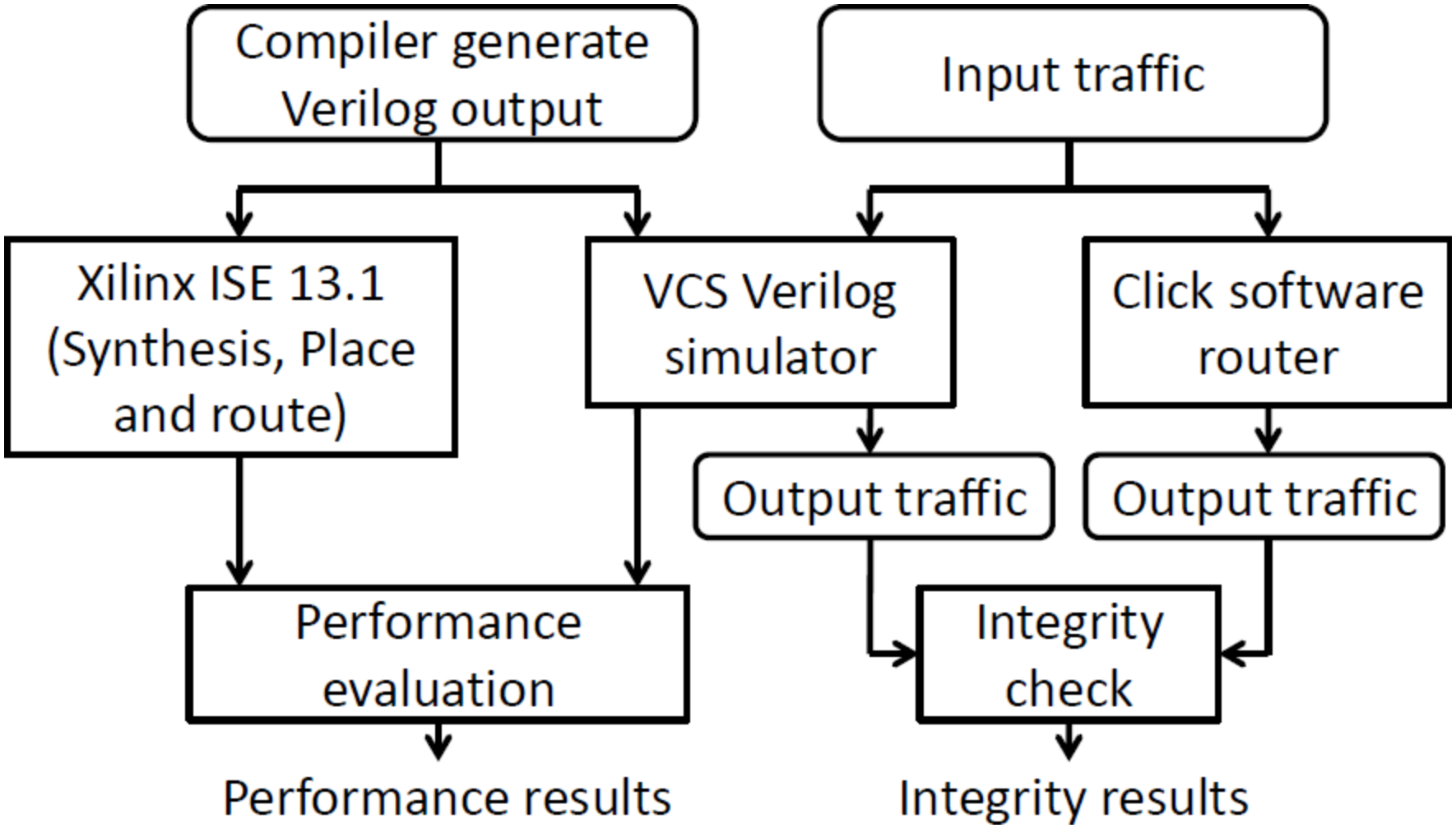
Engines	Threads	Virtex-6 LUT utilization%	Virtex-7 LUT utilization%
Four	One	27	28
Four	Two	35	34
Four	Four	50	48
Six	One	42	39
Six	Two	53	51
Six	Four	75	73

Gorilla and NetFPGA



- Eight clusters
- Netfpga system includes MACs for 4*10G Ethernet

Evaluation process – simulation part



Pin budgeting

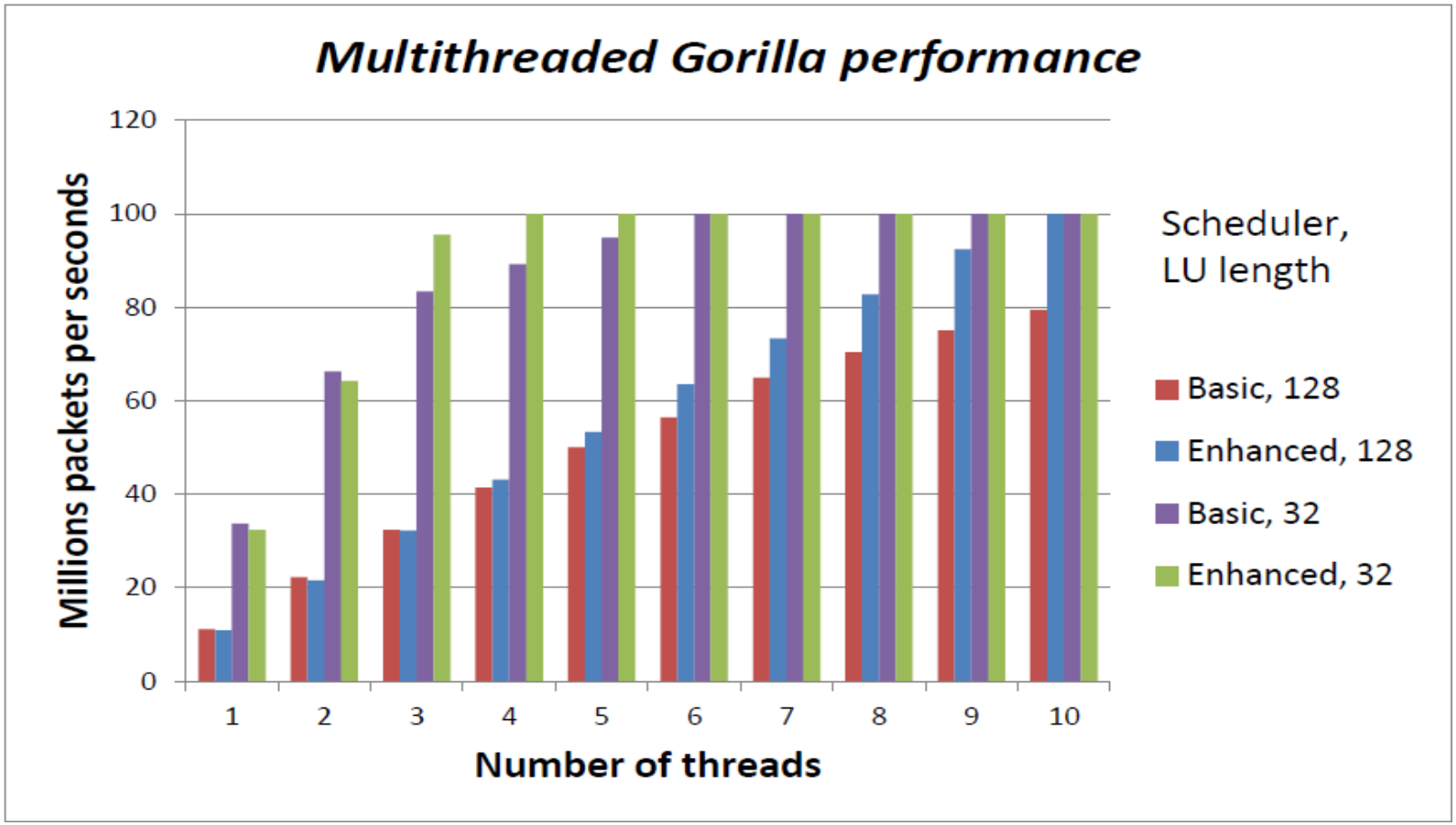
MPPS	Accelerator operation	QDRII chan-nels	QDRII+ chan-nels	Required MPLS traffic - QDRII	Required MPLS traffic - QDRII+
100	IPv4 LU	1	1	0	0
100	IPv6 LU	2	1	0	0
100	IPv4 LU-FC	3	2	0	0
100	IPv6 LU-FC	5*	3	27%	0
200	IPv4 LU	2	1	0	0
200	IPv6 LU	4	2	0	0
200	IPv4 LU-FC	6*	3	42%	0
200	IPv6 LU-FC	10*	5*	69%	27%

Configurations with * are not feasible in a single FPGA

Gorilla performance

- Architectural
  - Fast synchronization and customization between processing engines and accelerators
  - Maximize the utilization of scarce resources like (on-chip memory, pins)
- FPGA specific
  - Large parallelism and wide datapath to compensate low frequency
  - Flexibility in design to match the FPGA resource distribution
- Productivity
  - Template based design
    - Decoupling the functionality from performance related structures
    - Design space exploration for getting the area and timing closure

Thread scaling

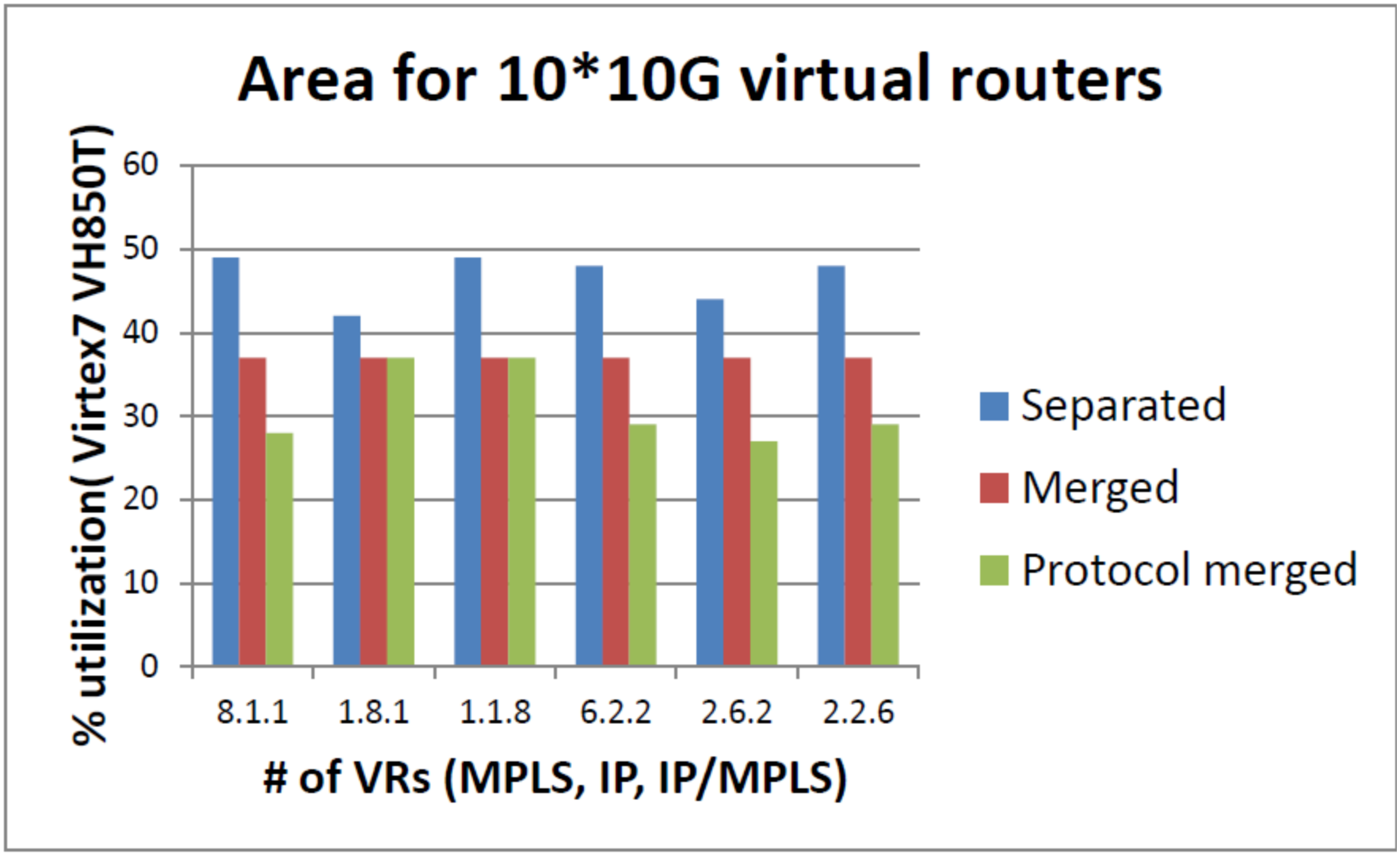


Eight clusters/ Four engines

Consolidating virtual routers

- When consolidation multiple routers with different protocols
  - Merged routers: All engines support all protocol
  - Isolated routers: Each router accommodate the necessary resources for its own protocol
- Merged routers is not fully efficient because we don't need all engines to be equipped to process all protocols
- Isolated routers duplicates the infrastructure resources
- We want to merge routers together when they have similar functionality and not merge routers with different functionality

Consolidating virtual routers - continue



IPv4 steps(simplified)

```
IPv4_check() {
  status = IPv4_header_integrity_check(Header);
  if (status == CHKSUM_OK)
    Next_step = IPv4_lookup;
  else
    Next_step = Exception;}

IPv4_lookup() {
  Da_class = lookupx.search(Header.IPv4_dstaddr);
  Sa_class = lookupy.search(Header.IPv4_srcaddr);
  if (Da_class == NOT_FOUND)
    Next_step = Exception;
  else if(Sa_class == NOT_FOUND)
    Next_step = Exception;
  else
    Next_step = IPv4_modify;}

IPv4_modify() {
  if((IP_update_fields(Header) == ZERO_TTL))
    Next_step = Exception;
  else {
    Dport = Da_class.dport;
    Next_step = Emit;}}
```

IPv4 accelerator interface

```
IPv4_check() { … }
IPv4_lookup() {
.
.
.
  Da_class = lookupx.search(Header.IPv4_dstaddr);
  Sa_class = lookupy.search(Header.IPv4_srcaddr);
.
.
.
Next_step = IPv4_QoS_count;}
IPv4_QoS_count() { … }
IPv4_update() { … }
```

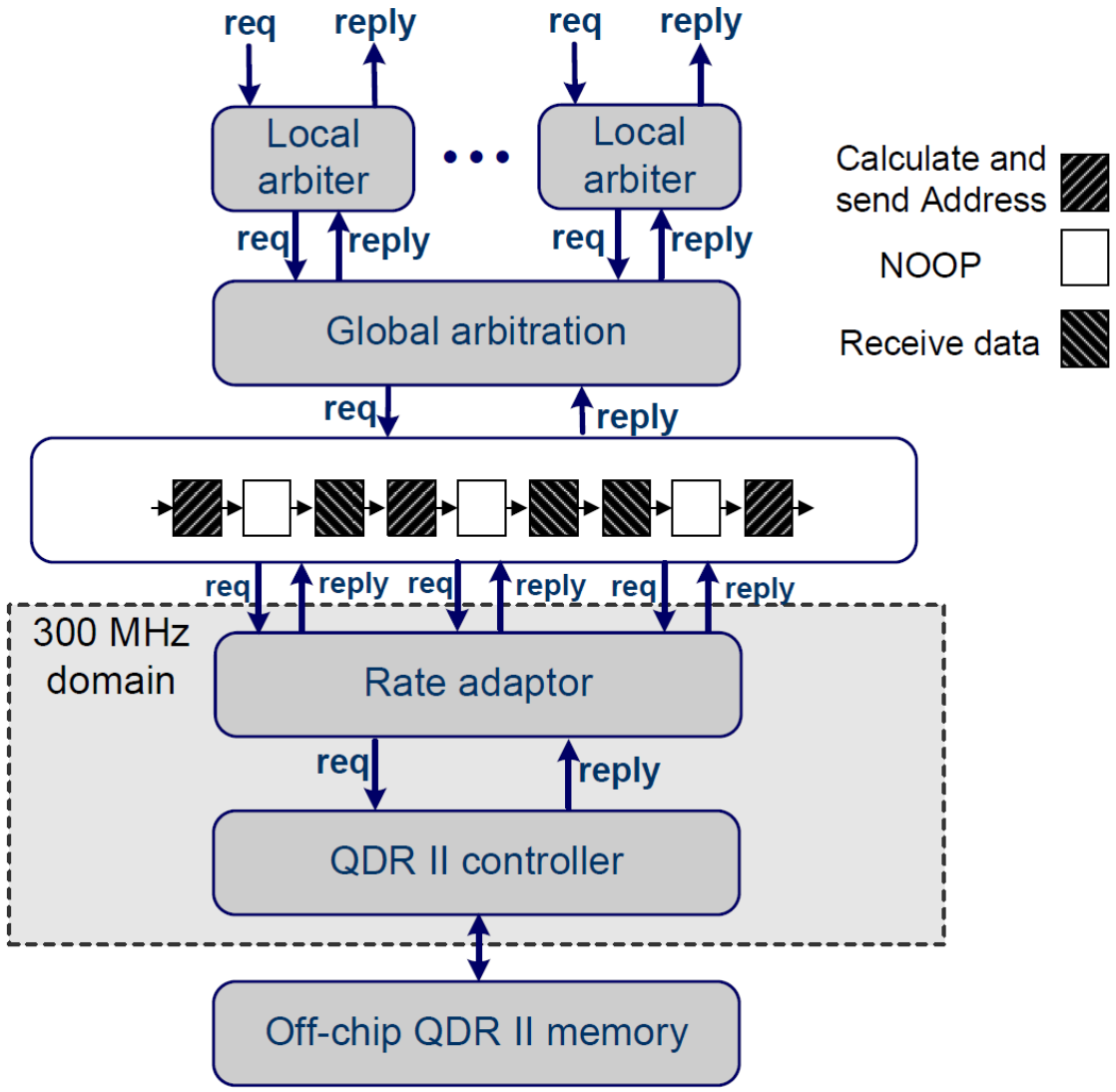
Lookup steps (simplified)

```
IPv4_lookup_first_level() {
    Found = 0;
    ip_address = Input;
    mem_address = Trie_address(1, 0, ip_address);
    trie_node = QDR_0.read(mem_address);
    Found = Is_leaf(trie_node);
    Next_step = IPv4_lookup_second_level;
}

IPv4_lookup_second_level() {
    Base_address = Chase_pointer(trie_node);
    mem_address = Trie_address(2, Base_address, ip_address);
    if (!Found)  trie_node = QDR_1.read(mem_address);
    Found = Is_leaf(trie_node);
    Next_step = IPv4_lookup_third_level;
}

IPv4_lookup_third_level() {
    Base_address = Chase_pointer(trie_node);
    mem_address = Trie_address(3, Base_address, ip_address);
    if (!Found) trie_node = QDR_2.read(mem_address);
    Output = Equivalent_forwarding_class(trie_node);
}
```

IPv4 lookup architecture



Major processing steps for packet processing

- Engines
  - IPv4 : 10 steps
  - IPv6 : 12 steps
  - MPLS : 13 steps
- Lookup
  - IPv4 : 3 steps - Three level trie using QDRII memory
  - IPv6 : 6 steps - Six level trie using QDRII memory
  - MPLS : 3 steps - Two level
    - First level is 4-way cache for second level, indexed with hash function
- QoS counting
  - 3*72 counters per packet: 6 steps
    - 3 QDR read operations and 3 QDR write operations per packet