Network Monitoring on Multi cores with Algorithmic Skeletons

Marco Danelutto, Luca Deri, Daniele De Sensi

Dept. of Computer Science, University of Pisa, Italy

PARCO 2011

Contents

Networking scenario

Increasing number of applications on IP and increasing speed of network interfaces (100M $\rightarrow$ 1G $\rightarrow$ 10G)

Networking scenario

Increasing number of applications on IP and increasing speed of network interfaces (100M $\rightarrow$ 1G $\rightarrow$ 10G)

Increasing need for highly efficient network monitoring applications

- special purpose hw/sw solutions from vendors

  e.g. Tilera multicores:
  - 64 to 100 cores per socket, cache only (private L1, local/shared L2, 4 external memory interfaces)
  - high speed network interfaces with direct cache packet injection
- *or* commodity processors with extremely efficient programming techniques
  - no unnecessary overheads with kernel interactions
  - no unnecessary overheads for synchronization

Processing scenario

- General purpose:
  - 6 to 8 full cores per socket
  - up to 64/128 threads per socket (Sun/Oracle T3/4)
  - 80 cores per socket already demonstrated (Intel Terascale prototype)

Processing scenario

- General purpose:
  - 6 to 8 full cores per socket
  - up to 64/128 threads per socket (Sun/Oracle T3/4)
  - 80 cores per socket already demonstrated (Intel Terascale prototype)
- Special purpose:
  - O(100) cores in GPUs
  - only suitable to support (some) data parallel code
  - impressive speedup over general purpose multicores: comparable speedup on a 48 AMD Magny chorus and on a (quite old) nVidia GTX285
  - time spent to send (packet) data to / receive (record) data from GPUs impairs usage for network monitoring

Current tools

- "low level" programming tools (Pthreads)
  $\rightarrow$ *full* responsibilities on programmers
- "higher level" programming tools (OpenMP, OpenCL)
  $\rightarrow$ *most* responsibilities still on programmers

Current tools

- "low level" programming tools (Pthreads)
  $\rightarrow$ *full* responsibilities on programmers
- "higher level" programming tools (OpenMP, OpenCL)
  $\rightarrow$ *most* responsibilities still on programmers

Recognized need for actually **high** level tools:

> *Architecting parallel software with design patterns, not just parallel programming languages. Our situation is similar to that found in other engineering disciplines where a new challenge emerges that requires a top-to-bottom rethinking of the entire engineering process;*

Asanovic et al. "A View of the Parallel Computing Landscape" CACM 2009

Parallel design patterns

- from sw engineering community
- introduced by Massingill, Mattson, Sanders in early 2000
  - "Patterns for parallel programming" Addison-Wesley 2004
- design patterns à la Gamma book
  - name, problem, solution, use cases, etc.
- define 4 pattern spaces (layered):
  concurrency, algorithms, implementation, mechanisms

Parallel design patterns

- from sw engineering community
- introduced by Massingill, Mattson, Sanders in early 2000
  - "Patterns for parallel programming" Addison-Wesley 2004
- design patterns à la Gamma book
  - name, problem, solution, use cases, etc.
- define 4 pattern spaces (layered):
  concurrency, algorithms, implementation, mechanisms

Application programmers

- should learn pattern lesson
- and implement it as needed in their own applications

Algorithmic skeletons

Independently developed but strictyl related to design patterns:

- from parallel programming community
- introduced by Cole in 1988 as
  - → parametric, reusable parallelism exploitation patterns
  - → directly exposed to programmers as language constructs/library calls
  - → completely hiding the technicalities related to parallelism exploitation
- languages & libraries since the '90
  - P3L, Skil, ASSIST, Muesli, SkeTo, Mallba, Muskel, Skipper, FastFlow, ...

Algorithmic skeletons

Independently developed but strictyl related to design patterns:

- from parallel programming community
- introduced by Cole in 1988 as
  - → parametric, reusable parallelism exploitation patterns
  - → directly exposed to programmers as language constructs/library calls
  - → completely hiding the technicalities related to parallelism exploitation
- languages & libraries since the '90
  - P3L, Skil, ASSIST, Muesli, SkeTo, Mallba, Muskel, Skipper, FastFlow, ...

Application programmers

- instantiate existing skeletons
- to (safely and efficiently) build their parallel application

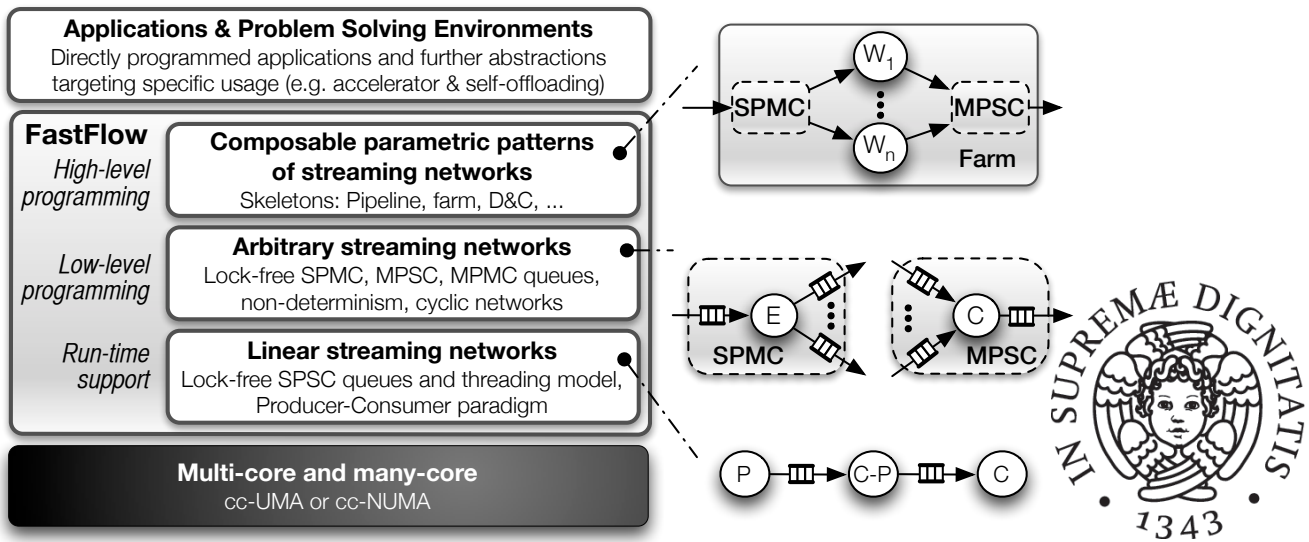
Main goal of this work

- exploit structured parallel programming techniques
- to support network monitoring
- on commodity hardware

FastFlow

Advanced programming framework

- targeting multicores
- minimizing synchronization latencies
- streaming support through skeletons
- expandable
- open source



FastFlow: simple streaming networks

Single Producer Single Consumer (SPSC) queue

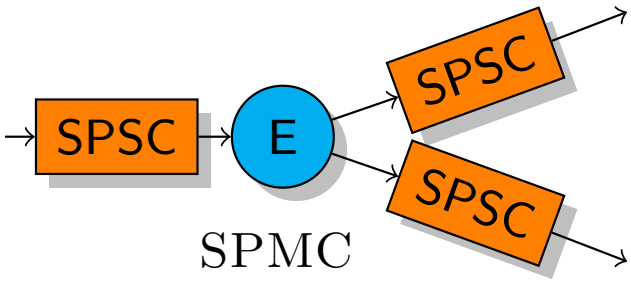
- uses results from the '80s
- lock-free, wait-free
- no memory barriers for Total Store Order processor (e.g. Intel, AMD)
- single memory barrier for weaker memory consistency models (e.g. PowerPC)

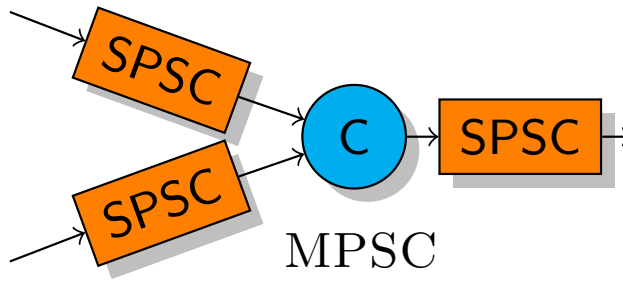
→ very low latency in communications

FastFlow: simple streaming networks

Other queues: SPMC MPSC MPSC

- one-to-many, many-to-one and many-to-many synchronization and data flow
- use an explicit arbiter thread
- providing lock-free and wait-free arbitrary data-flow graphs
- cyclic graphs (provably deadlock-free)

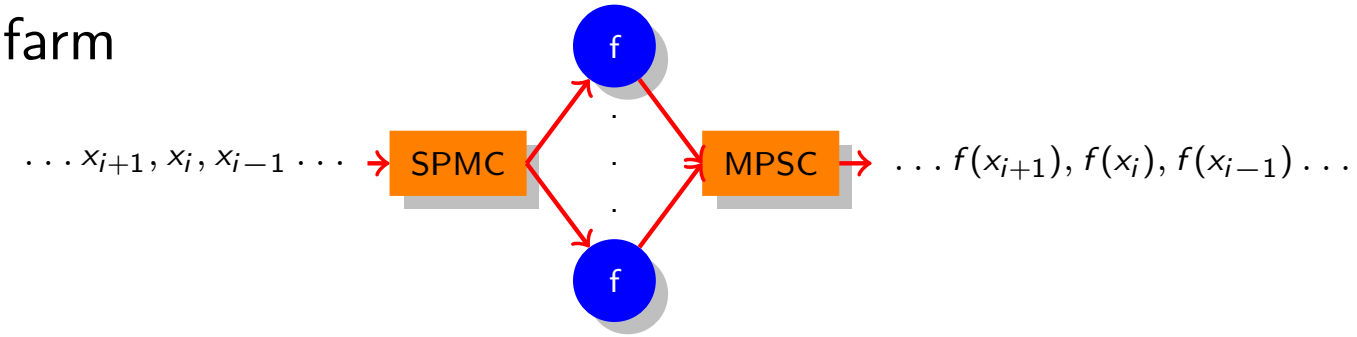




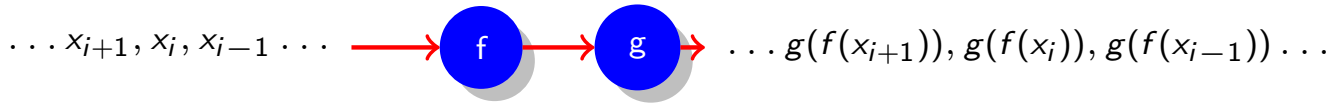
FastFlow: high level programming abstractions

Several "streaming" skeletons provided

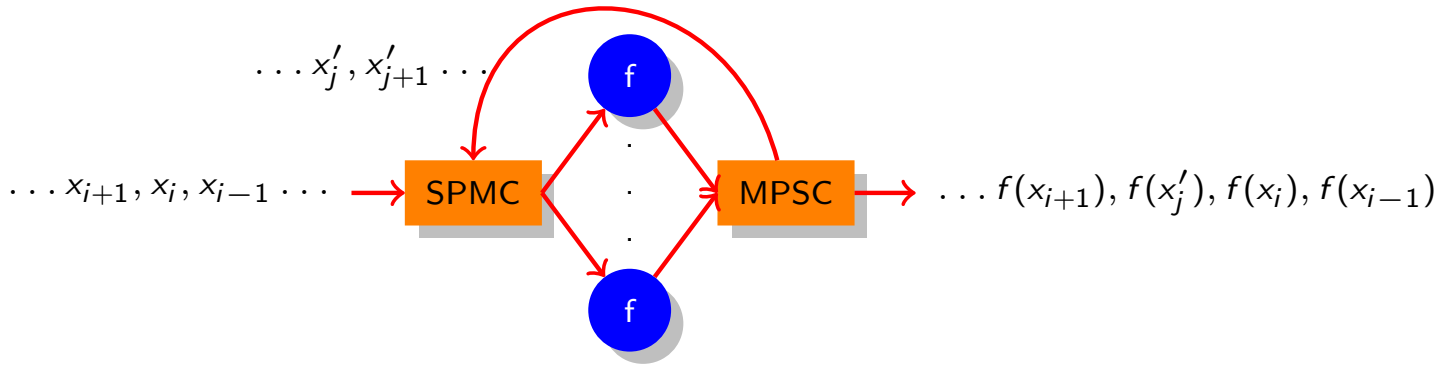
- farm



- pipeline



- farm with feedback (divide & conquer)



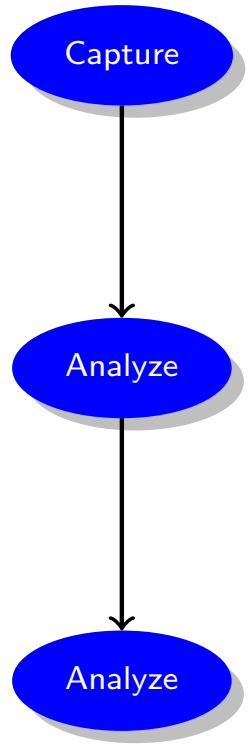
Sample code

```
int main(int argc, char * argv[]) {
    ...
    ff_pipeline pipe;

    s1 = new PacketCaptureStage(Npackets);
    s2 = new PacketAnalysisStage(...);
    s3 = new PacketAnalysisStage(...);

    pipe.add_stage(s1);
    pipe.add_stage(s2);
    pipe.add_stage(s3);

    if (pipe.run_and_wait_end()<0) {
        // handle error ...
    }
    return 0;
}
```



FastFlow results

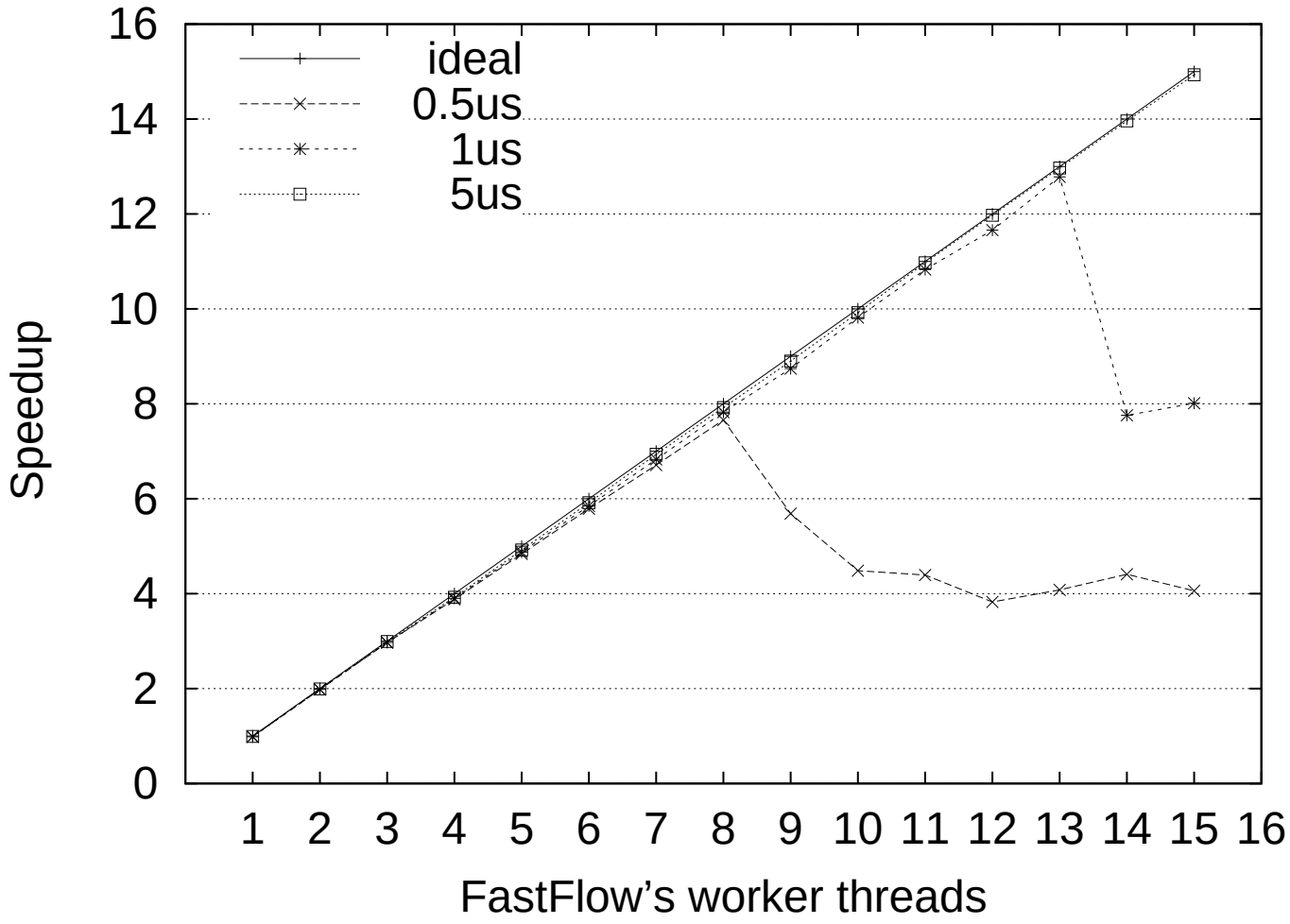
Benchmark	Parameters	Skeleton used	Speedup / #cores
Matrix Mult.	1024x1024	farm no collector	7.6 / 8
Quicksort	50M integers	D&C	6.8 / 8
Fibonacci	Fib(50)	D&C	9.21 / 8

Table: Microbenchmarks parallelized using FastFlow.

Application	Skeleton used	Performance
YaDT-FF	D&C	4.5-7.5 Speedup
StochKit-FF	farm	10-11 Scalabitily
SWPS3-FF	farm no collector	12.5-34.5 GCUPS

Table: Applications parallelized using FastFlow.

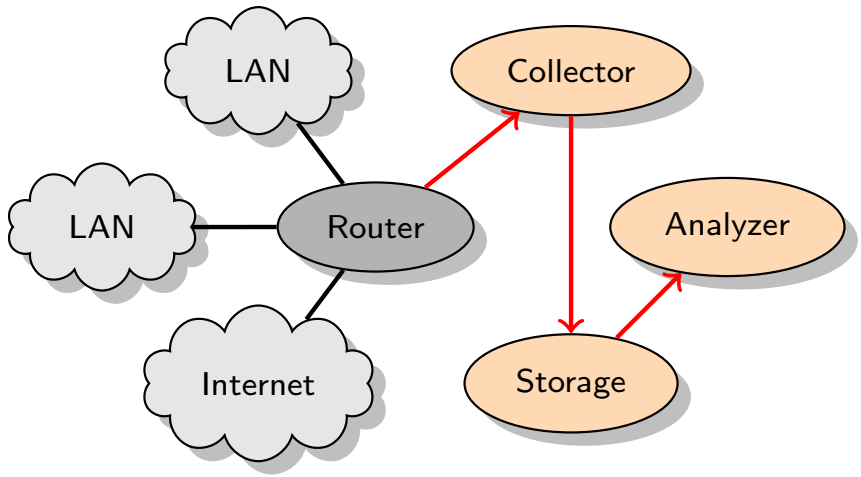
FastFlow results



NetFlow

Network protocol to collect IP traffic information

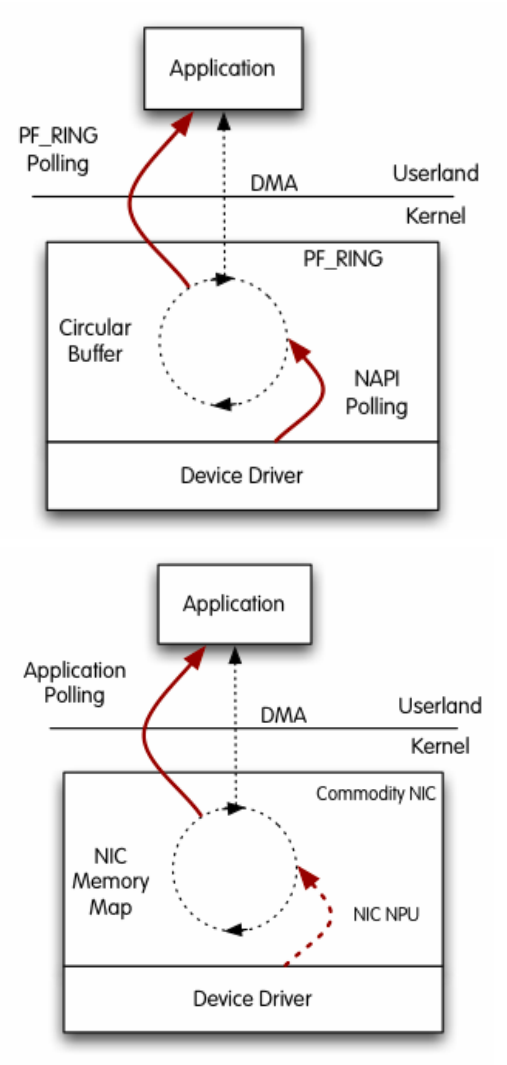
- by Cisco
- de facto standard ($\rightarrow$ IPFIX)
- works on *flows*: unidir sequence of packets with same source, dest and type of protocol
- generates records hosting:
  - version and sequence number, timestamps
  - layer 3 headers & routing info



PF_RING

Linux new type of network socket

- extremely efficient device to kernel ring packet copy
- exploits Linux NAPI interface (interrupts + polling)
- two operation modes
  - device to (multiple) kernel rings
    → supports packet directing to different applications
  - device to memory mapped kernel ring
    → zeroes copy time
    → but directs packets to one application only



Architectural design

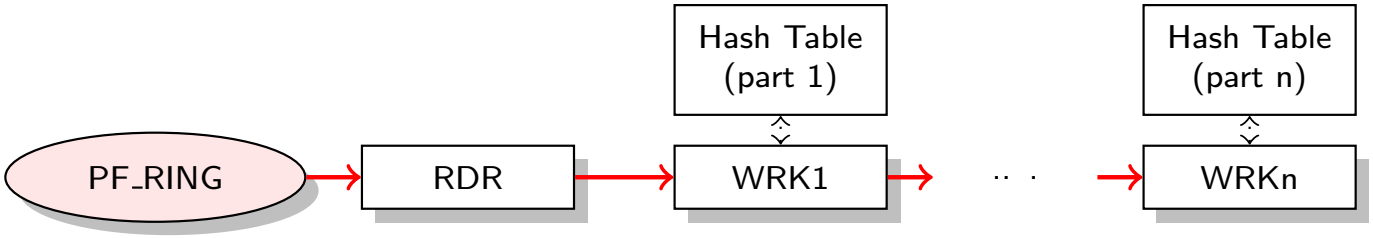
Three different kind of parallel designs

- simple multiple design re-using sequential components
- explore different possibilities
- to match packet capture related constrains

General design:

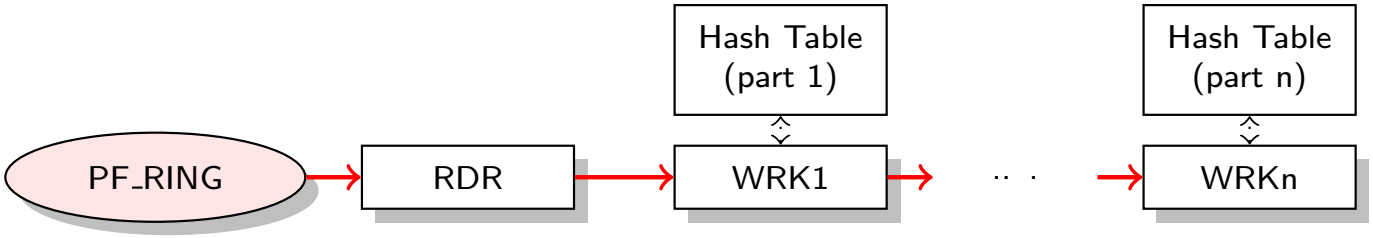
- modular in the number of packet capture queues
  - → one or more `PF_RING` queues
- modular in the number of threads processing incoming packets
  - → stages in a pipeline: each stage processes part of the captured packets
  - → more pipelines attached to different `PF_RING`

Base design



pipe: 1 `PF_RING` queue, 1 reader stage, *n* stages processing packets

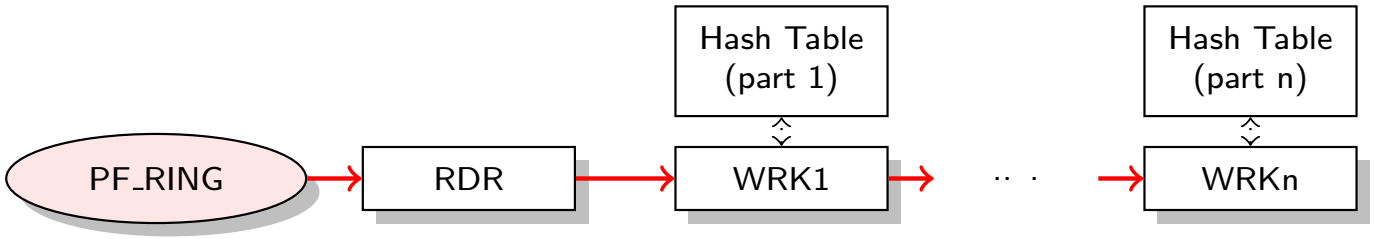
Base design



pipe: 1 `PF_RING` queue, 1 reader stage, *n* stages processing packets

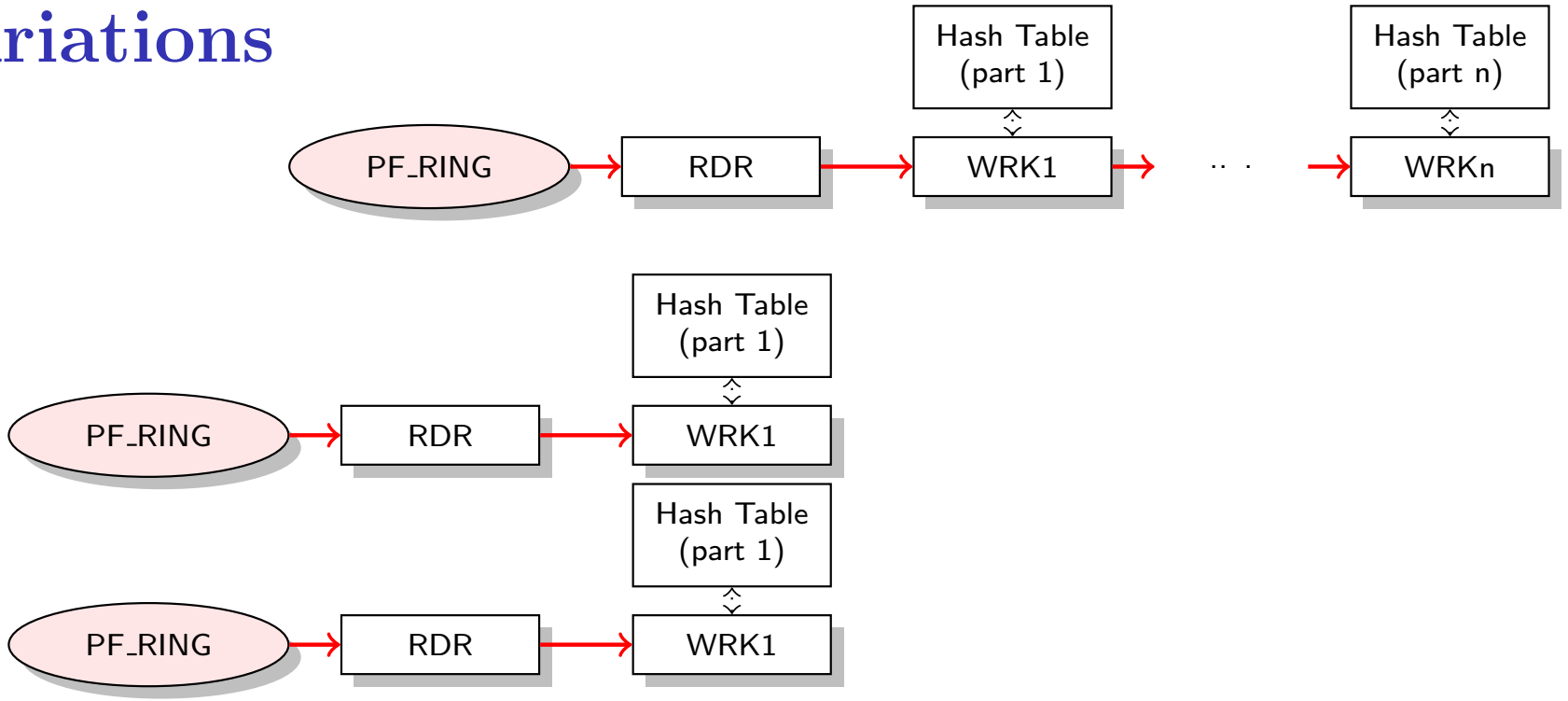
- *RDR*
  - reads captured packets
  - groups them in messages
  - each packet directed to one stage through hash label
  - forwards messages through the pipeline
- *WRK*
  - reads a message (group of packets)
  - processes packets with proper (own) hash flag
  - on termination forwards resulting records

Variations


Hash Table
(part 1)
Hash Table
(part n)
PF_RING
RDR
WRK1
WRKn

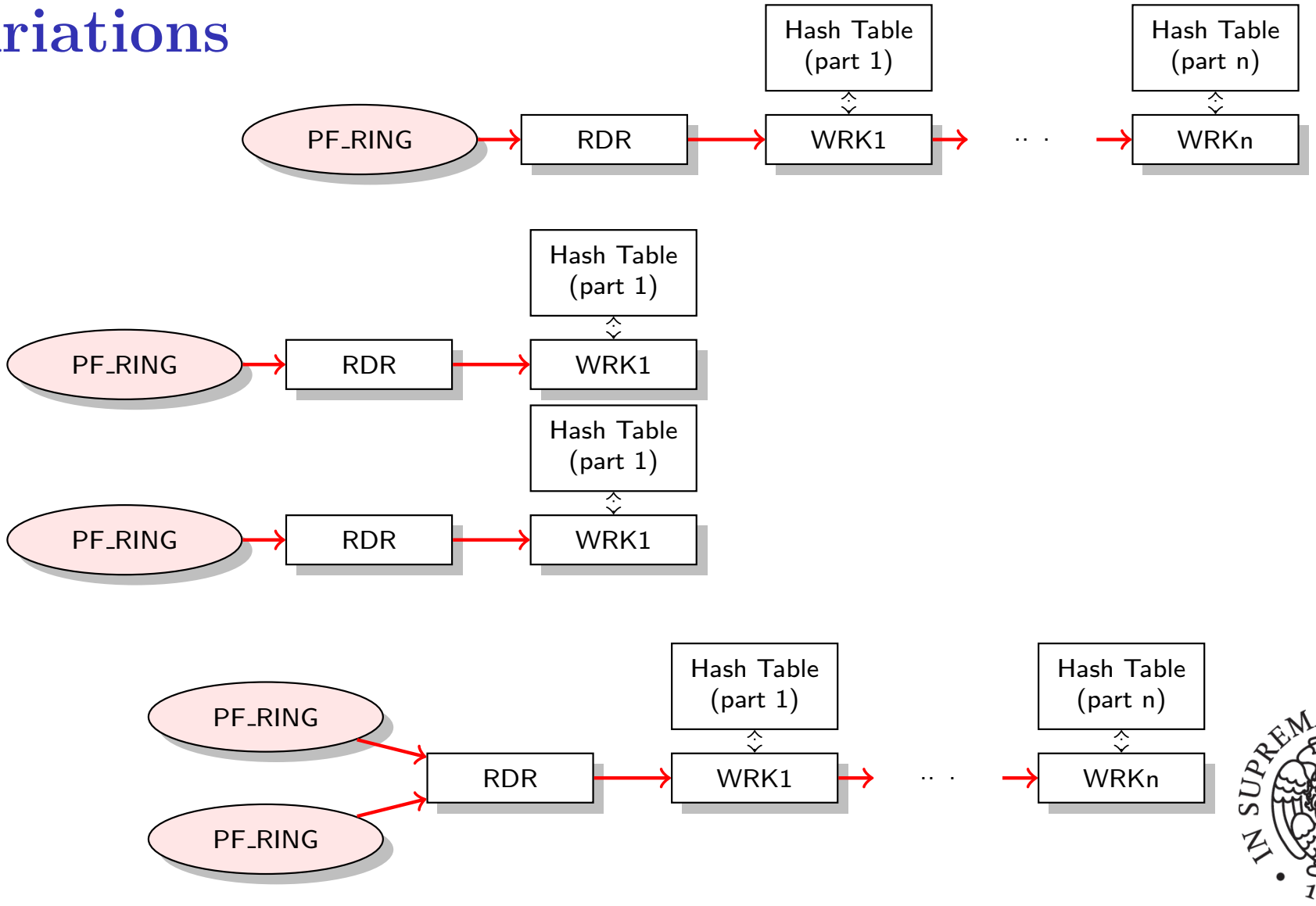

IN SUPREMÆ DIGNITATIS
1343

Variations


Hash Table (part 1)
Hash Table (part n)
PF_RING
RDR
WRK1
WRKn
Hash Table (part 1)
PF_RING
RDR
WRK1
Hash Table (part 1)
PF_RING
RDR
WRK1


IN SUPREMÆ DIGNITATIS
1343

Variations


Hash Table (part 1)
Hash Table (part n)
PF_RING
RDR
WRK1
WRKn
Hash Table (part 1)
PF_RING
RDR
WRK1
Hash Table (part 1)
PF_RING
RDR
WRK1
Hash Table (part 1)
Hash Table (part n)
PF_RING
PF_RING
RDR
WRK1
WRKn

Packet processing

- Basic network monitoring → very fine grain processing
  - extract simple data fields from packets
- More data processing needed for more evolved inspection strategies
- ffProbe:
  - experiments made with the finer grain processing functions
    → results improve with larger grain
    and with more stages / pipelines

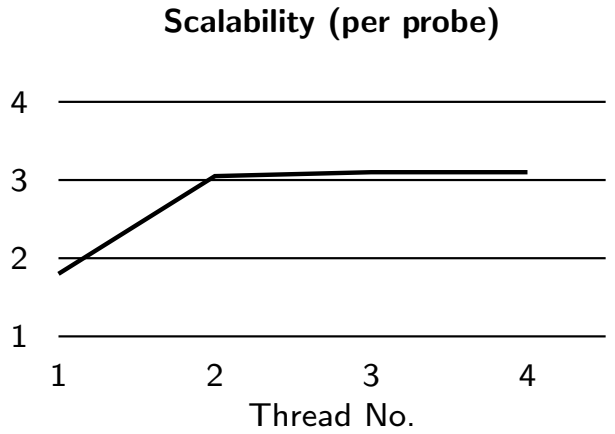
Experimental results (absolute)

schema	Par. degree	Thr#	Mega PPS	Memory footprint
1 pipeline	Seq	1	3.76	50M
n workers	1R + 1W	2	6.45	94M
	1R + 2W	3	8.42	78M
2 PF_RING	1R + 1W per pipe	4	10.150	64M
2 pipelines	1R + 2W per pipe	6	10.143	64M
2 PF_RING	2R + 1W	3	5.54	115M
1 pipeline	2R + 2W	4	9.13	150M
	2R + 3W	5	10.033	150M

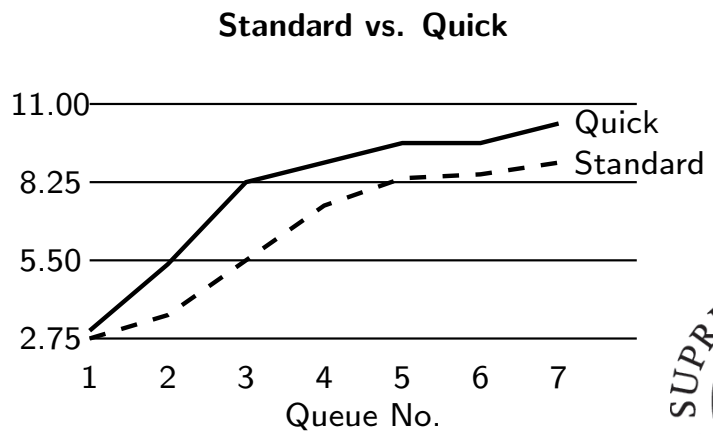
on a Dual Nehalem (Xeon E5520, 2.27GHz) with 10 Gbit Intel-based Silicom NICs

Comparisons (nProbe)

A single instance does not scale with the thread number over 3Mpps:



Multiple instances on different PF_RING queues process as many packets as ffProbe:



Assessments

- High level, highly efficient parallel programming environment
  - supports network monitoring
  - on commodity hardware
  - targeting 10Gbps network interfaces
  - different parallel design experimented with negligible programming effort (once base sequential components have been defined)

Assessments

- High level, highly efficient parallel programming environment
  - supports network monitoring
  - on commodity hardware
  - targeting 10Gbps network interfaces
  - different parallel design experimented with negligible programming effort (once base sequential components have been defined)
- High speed network monitoring
  - special purpose hw $\rightarrow$ commodity hw
  - systems (as tested) in the 3-4K euro range

Future (ongoing) work

To be improved:

- modularization of analysis code (plugin)
- ...

Product design currently on going:

- clean up and engineering of the code
- documentation (internal, user)
- experiments on larger core configurations
- to be released under open source license

Any questions ?

{marcod,desensi}@di.unipi.it, deri@ntop.org