Huge pages impact analysis on high-speed network packet capture in the Linux kernel

Master's thesis

Computer and Information Engineering

Laura Trivelloni

Advisor: Prof. Marco Cesati

Co-Advisors: Eng. Emiliano Betti
Eng. Pierpaolo Santucci

April 15, 2019

Introduction

Scenario

Packet capture

- traffic analysis
- network monitoring

Requirements

- high-speed
- low packet loss rate

Scenario

Packet capture

- traffic analysis
- network monitoring

Requirements

- high-speed
- low packet loss rate

10–40 Gigabit Ethernet

The Linux kernel

Linux is an **open-source** operating system.

Frameworks for high-performance packet capture:

- **PFQ**
- **Intel DPDK**
- **PF_RING ZC**

PFQ Framework

PFQ I/O

- highly optimized for **multi-core** architecture
- RX and TX **line-rate on 10-Gbit links**
- user-kernel memory-mapped buffer backed with **Huge Pages**

Translation Lookaside Buffer

Virtual memory:

- software-controlled set of memory addresses
- data **protection** and **control** among processes

Translations from virtual to physical address cost CPU cycles.

Translation Lookaside Buffer

Virtual memory:

- software-controlled set of memory addresses
- data **protection** and **control** among processes

Translations from virtual to physical address cost CPU cycles.

→ cache for resolved addresses:

Translation Lookaside Buffer

Translation Lookaside Buffer

Virtual memory:

- software-controlled set of memory addresses
- data **protection** and **control** among processes

Translations from virtual to physical address cost CPU cycles.

$\rightarrow$ cache for resolved addresses:

Translation Lookaside Buffer

Not always enough to save CPU clocks:

- TLB miss event

Huge pages

How to reduce the CPU cycles spent for translations?

- fewer translations $\rightarrow$ fewer CPU cycles
- **wider address range** for each TLB entry
- the same number of TLB entry covers **more addresses**
- more likely TLB **hits** events

Huge pages

How to reduce the CPU cycles spent for translations?

- fewer translations → fewer CPU cycles
- **wider address range** for each TLB entry
- the same number of TLB entry covers **more addresses**
- more likely TLB **hits** events

Huge Pages

4K → 2M

Linux support for huge pages

Transparent Huge Pages

- **on-demand** allocation
- without user awareness

Really available?

HugeTLB filesystem

- **pre-allocated** pages
- **application awareness**

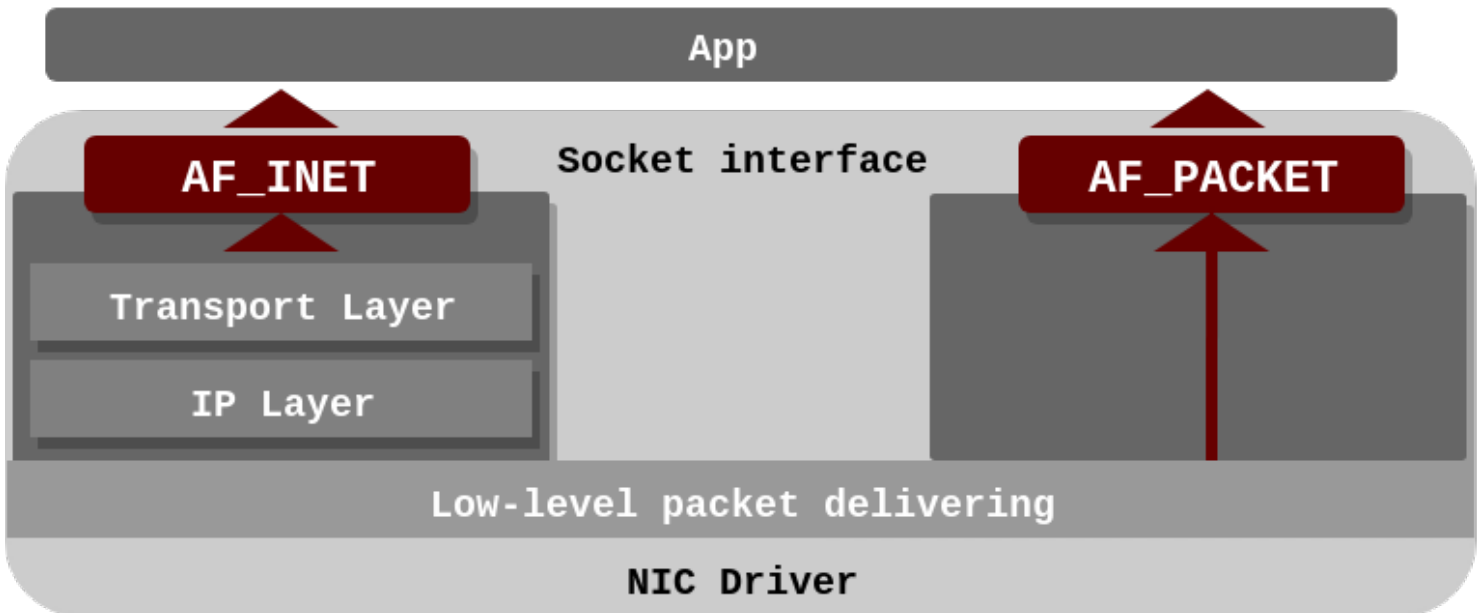
Not *transparent!*

Problem analysis

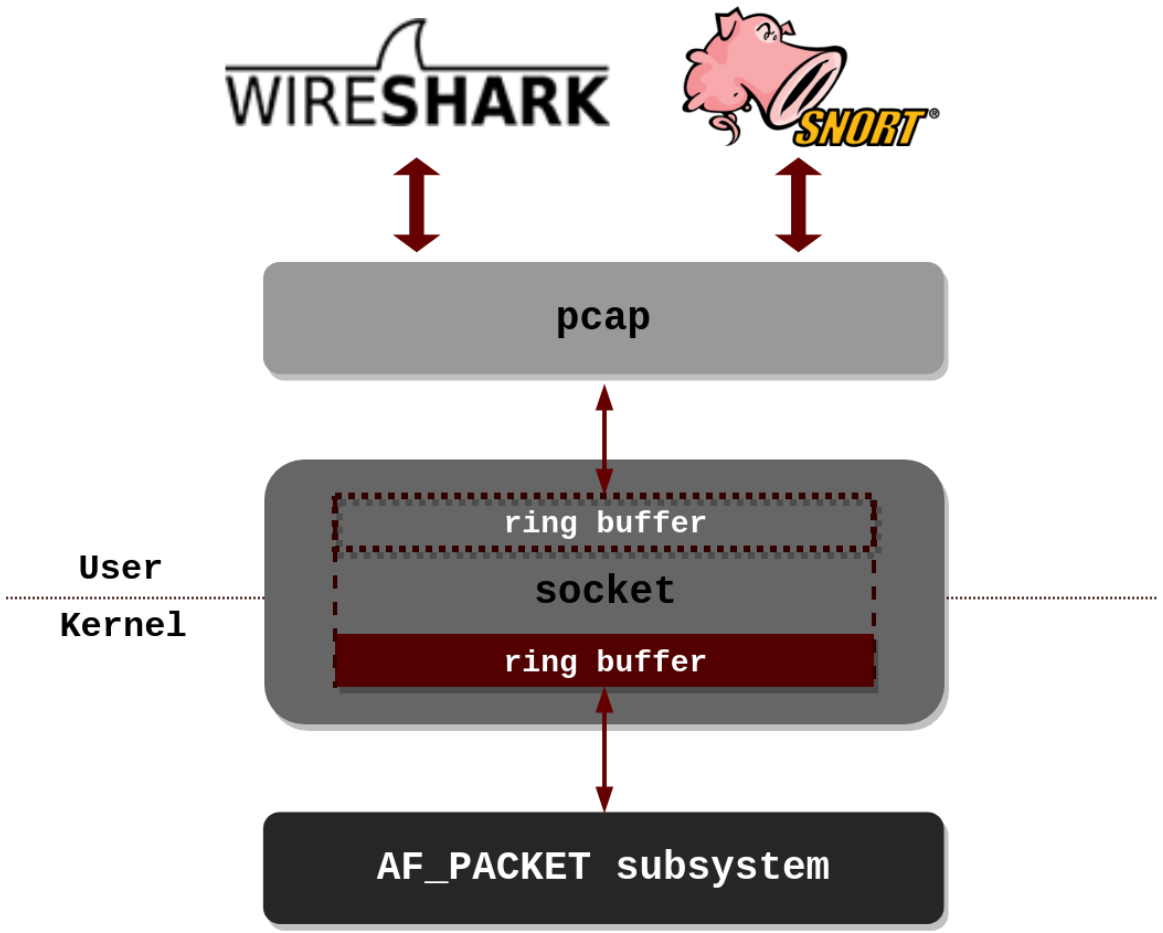
The AF_PACKET kernel subsystem

The `AF_PACKET` implements the networking domain of the raw sockets:

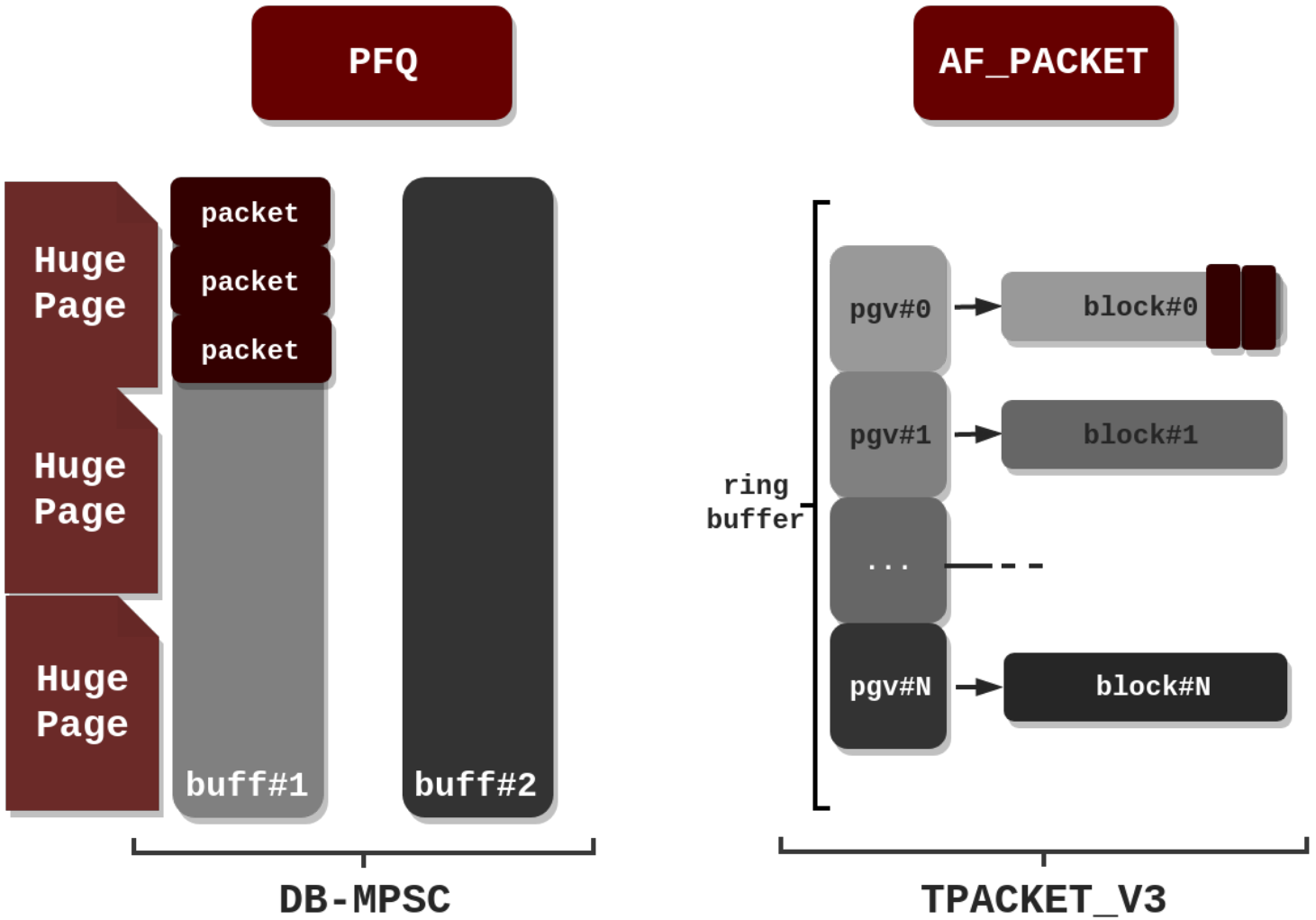
- low-level access to the network interface



Traffic analysis in Linux



PFQ vs AF_PACKET



PFQ vs AF_PACKET

Premises:

- the Linux kernel **does not use Huge Pages** in the buffer reception
- PFQ that optimizes the packet capture **uses Huge Pages**

PFQ vs AF_PACKET

Premises:

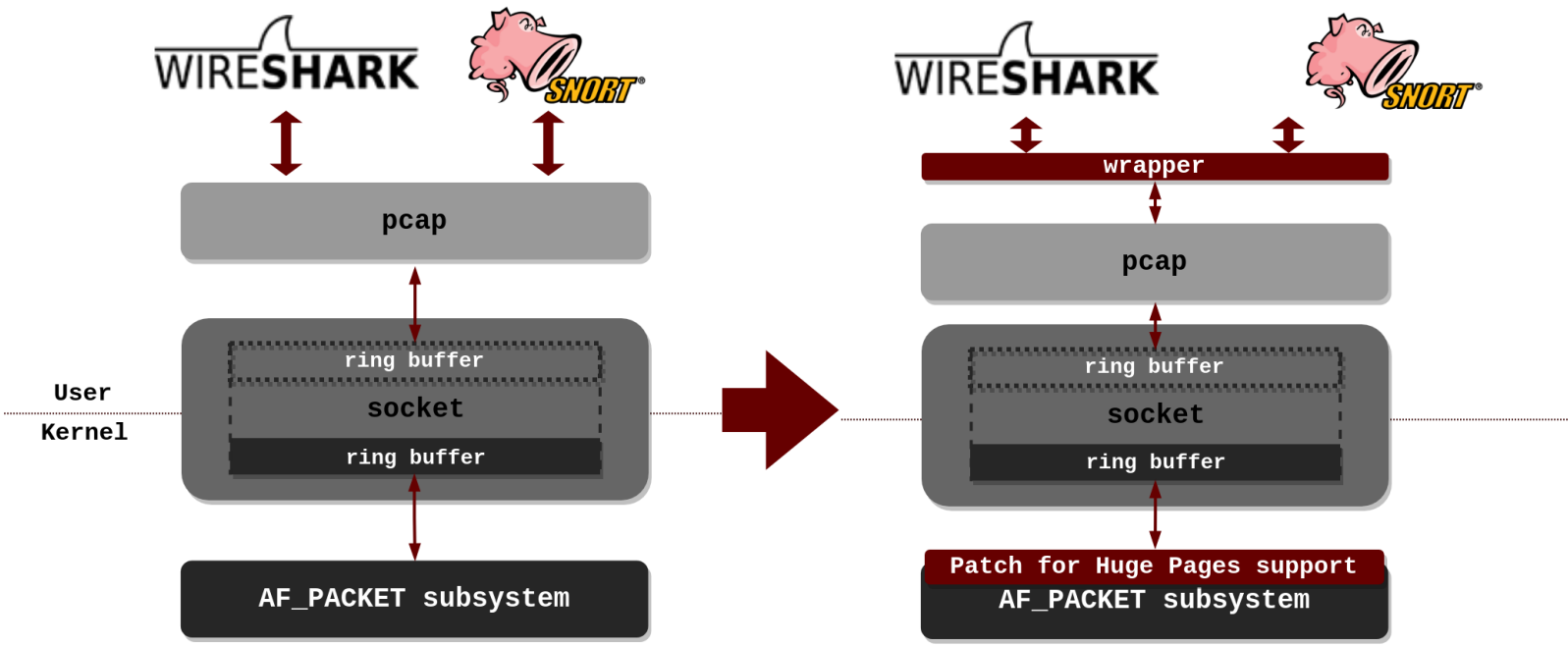
- the Linux kernel **does not use Huge Pages** in the buffer reception
- PFQ that optimizes the packet capture **uses Huge Pages**

Our question

Can make sense introducing a buffer allocation backed with the Huge Pages in the AF_PACKET domain?

Proposed solution

Our solution



Interface

The solution is exported in user space as a socket configuration option.

Application awareness implies:

- easy testing activities

Interface

The solution is exported in user space as a socket configuration option.

Application awareness implies:

- easy testing activities
- to reach compatibility with existing libraries

Interface

The solution is exported in user space as a socket configuration option.

Application awareness implies:

- easy testing activities
- to reach compatibility with existing libraries
  - `pcap`

Current block allocation policy

Attempts of block allocation in order of preference:

1. physically contiguous **direct** memory request
2. only virtually contiguous memory request
3. physically contiguous forced memory request

New block allocation policies

One allocation policy among the following can be forced:

1. memory backed with *pre-allocated huge pages*

New block allocation policies

One allocation policy among the following can be forced:

1. memory backed with *pre-allocated huge pages*
   - HugeTLB filesystem

New block allocation policies

One allocation policy among the following can be forced:

1. memory backed with *pre-allocated huge pages*
   - HugeTLB filesystem
2. memory backed with *on-demand huge pages*

New block allocation policies

One allocation policy among the following can be forced:

1. memory backed with *pre-allocated huge pages*
   - HugeTLB filesystem
2. memory backed with *on-demand huge pages*
   - Transparent Huge Pages

New block allocation policies

One allocation policy among the following can be forced:

1. memory backed with *pre-allocated huge pages*
   - HugeTLB filesystem
2. memory backed with *on-demand huge pages*
   - Transparent Huge Pages
3. physically contiguous direct memory request

New block allocation policies

One allocation policy among the following can be forced:

1. memory backed with *pre-allocated huge pages*
   - HugeTLB filesystem
2. memory backed with *on-demand huge pages*
   - Transparent Huge Pages
3. physically contiguous direct memory request
4. only virtually contiguous memory request

New block allocation policies

One allocation policy among the following can be forced:

1. memory backed with *pre-allocated huge pages*
   - HugeTLB filesystem
2. memory backed with *on-demand huge pages*
   - Transparent Huge Pages
3. physically contiguous direct memory request
4. only virtually contiguous memory request
5. physically contiguous forced memory request

New block allocation policies

One allocation policy among the following can be forced:

1. memory backed with *pre-allocated huge pages*
   - HugeTLB filesystem
2. memory backed with *on-demand huge pages*
   - Transparent Huge Pages
3. physically contiguous direct memory request
4. only virtually contiguous memory request
5. physically contiguous forced memory request

If not specified, the original allocation policy is performed.

Integration with the `pcap` library

Making cascaded modifications to a library is a costly operation.

- exploiting library **dynamic linking**

Integration with the `pcap` library

Making cascaded modifications to a library is a costly operation.

- exploiting library **dynamic linking**
- **alternative routines** corresponding to the orginal symbols at compile-time

Integration with the `pcap` library

Making cascaded modifications to a library is a costly operation.

- exploiting library **dynamic linking**
- **alternative routines** corresponding to the orginal symbols at compile-time
- using **environment variables** to transfer not supported information

Integration with the `pcap` library

Making cascaded modifications to a library is a costly operation.

- exploiting library **dynamic linking**
- **alternative routines** corresponding to the orginal symbols at compile-time
- using **environment variables** to transfer not supported information

No need to recompile the pcap library!

Results

Test environment - sender side

- generated synthethic traffic

Test environment - sender side

- generated synthethic traffic
  - pktgen kernel tool

Test environment - sender side

- generated synthethic traffic
  - pktgen kernel tool
- traffic saved as a `pcap` file

Test environment - sender side

- generated synthethic traffic
  - pktgen kernel tool
- traffic saved as a `pcap` file
  - tcpdump tool

Test environment - sender side

- generated synthethic traffic
  - pktgen kernel tool
- traffic saved as a `pcap` file
  - tcpdump tool
- `pcap` trace sent using tcpreplay tool

Test environment - sender side

- generated synthethic traffic
  - pktgen kernel tool
- traffic saved as a `pcap` file
  - tcpdump tool
- `pcap` trace sent using tcpreplay tool
- rate at 10 Gb/s

Test environment - receiver side

- hash computation on each received packet content

Test environment - receiver side

- hash computation on each received packet content
  - traffic analysis simulation

Test environment - receiver side

- hash computation on each received packet content
  - traffic analysis simulation
- variable delay factor every 100 packets

Test environment - receiver side

- hash computation on each received packet content
  - traffic analysis simulation
- variable delay factor every 100 packets
  - 1000 multiplications repeated for *delay factor* times

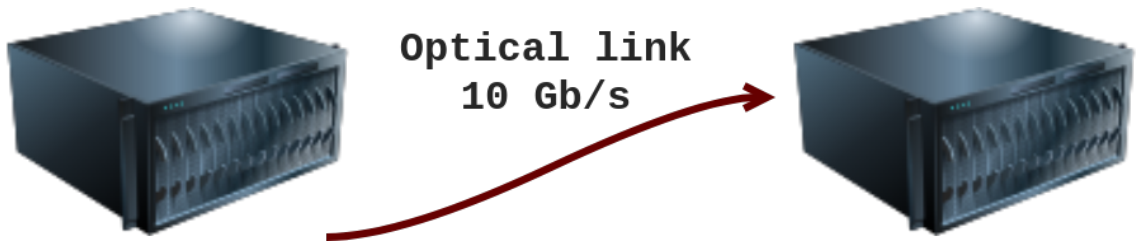
Test environment - receiver side

- hash computation on each received packet content
  - traffic analysis simulation
- variable delay factor every 100 packets
  - 1000 multiplications repeated for *delay factor* times
  - buffer overloading to simulate traffic analysis peaks

Test environment - receiver side

- hash computation on each received packet content
  - traffic analysis simulation
- variable delay factor every 100 packets
  - 1000 multiplications repeated for *delay factor* times
  - buffer overloading to simulate traffic analysis peaks
- statistics computed on 10 runs for each configuration

Test machines



Packet generator

- CPU: Intel® Xeon E5
- NIC: Intel® 82599ES

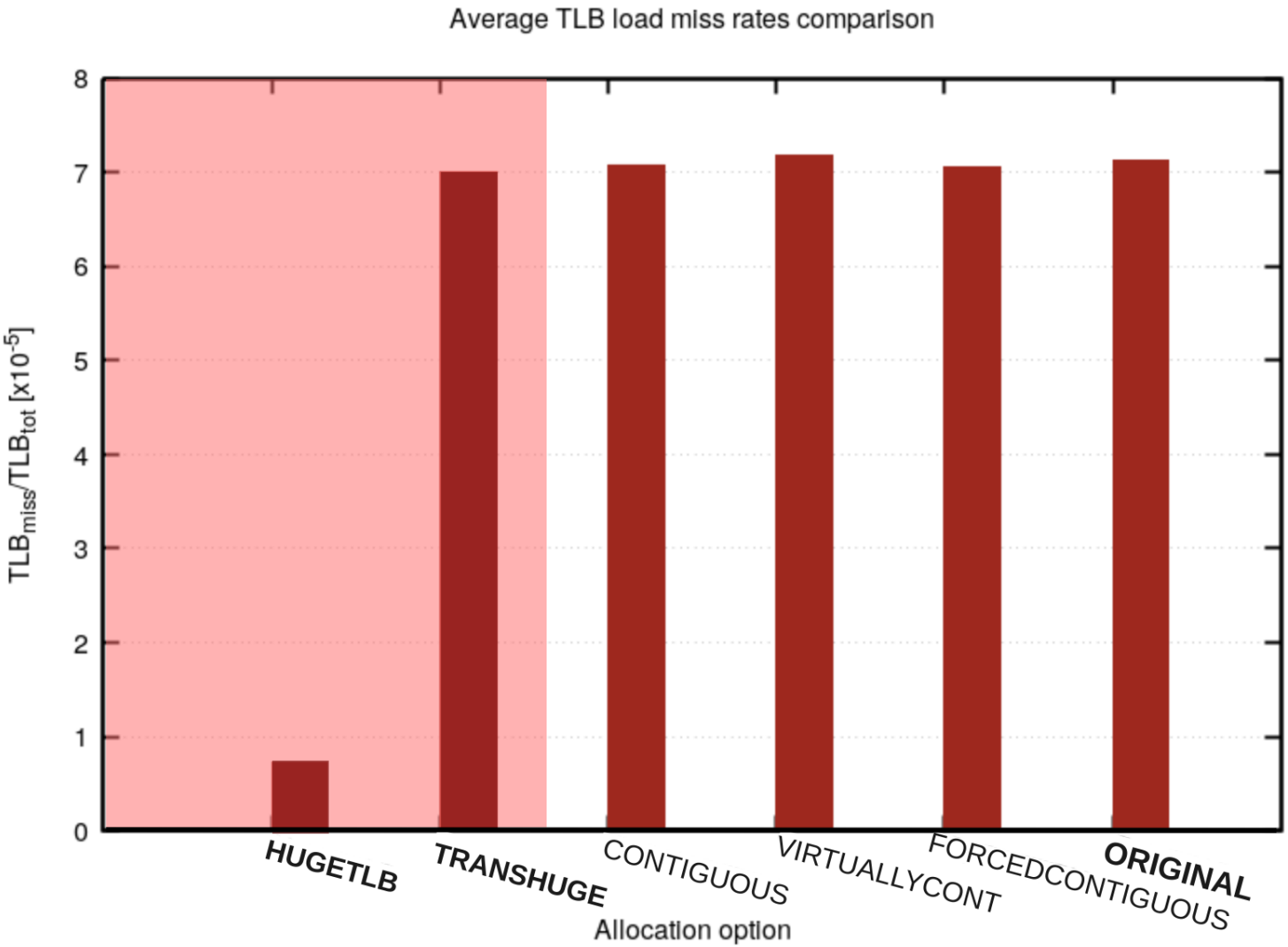
Packet receiver

- CPU: Intel® Xeon E5
- NIC: Intel® X710

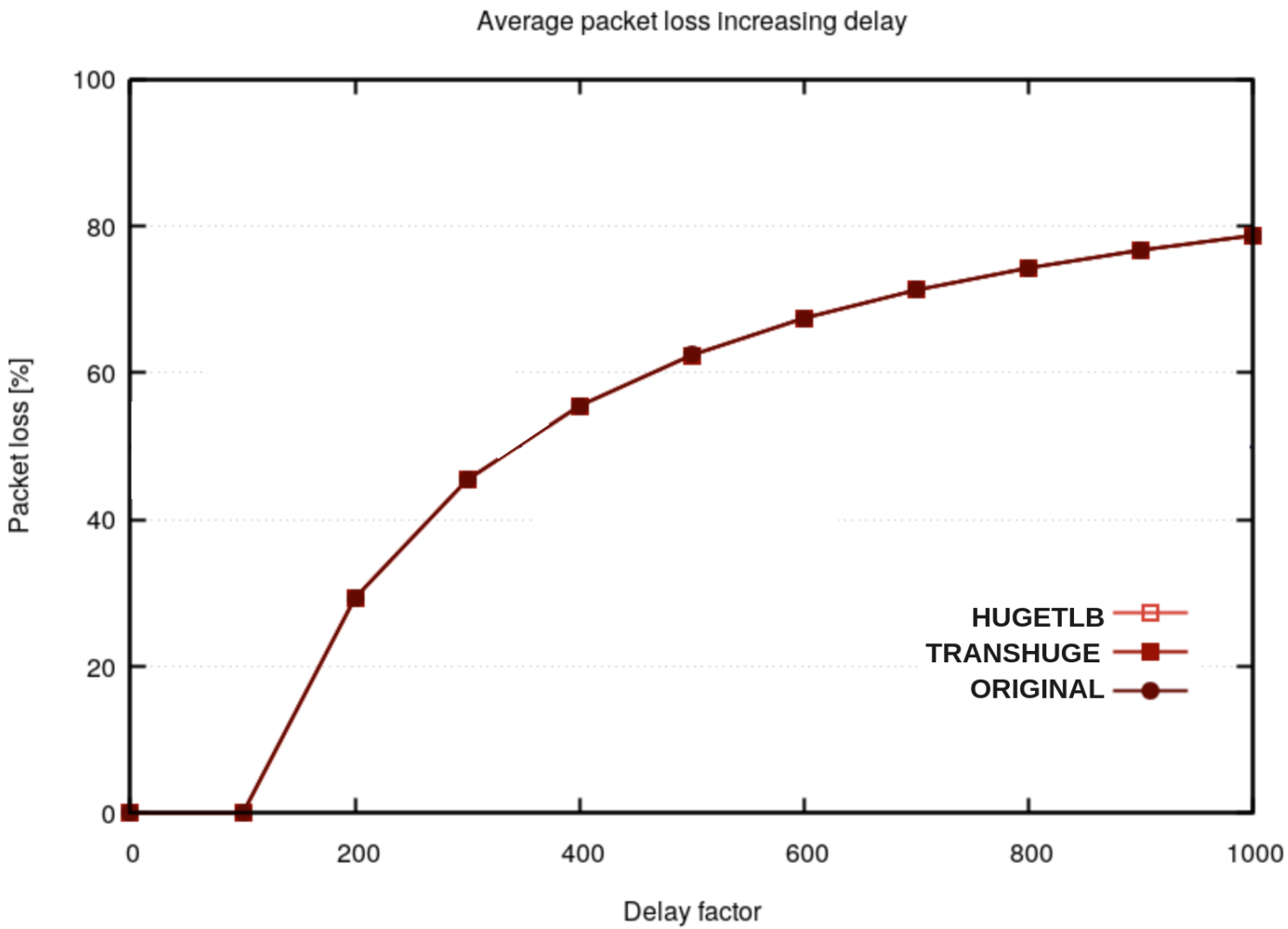
Performance measurements

- **Performance Measurement Unit** provides hardware-based counters
- **perf** tool analyser to measure events:
  - **CPU cycles**
  - **TLB accesses**
  - **TLB misses**
- **Time Stamp Counter** register
  - **CPU cycles** counter in Intel x86

TLB load miss rates



Packet loss



Conclusions

Conclusions

Impact analysis of two new allocation policies based on **huge pages**:

- **compatibility** with widespread applications
- effective **improvement** in terms of TLB miss rates
- **no gain** in terms of packet loss rate

Thank You for listening!

Laura Trivelloni