【Internet Week 2016】

パケットフォワーディングを支える技術2016

ソフトウェアによるパケット処理の話 ～ソフトウェアルータからNFVまで～

東京大学大学院情報理工学系研究科
特任助教 浅井大史
<panda@hongo.wide.ad.jp>

2016年11月29日

今日の話

- データプレーンのソフトウェア技術：ソフトウェアルータ, NFV
  ≠コントロールプレーン: SDN, White box switch
  - ソフトウェアによる高速パケット処理技術
    ～ハードウェアアーキテクチャを知る～
  - NFVに向けて
    ～仮想化を知る～

このあたりの疑問にお答えします

- 高速データプレーン技術(e.g., DPDK)は何故速い？
- 割込みは重いって本当？
- VMでVNFを動かすと仮想化のオーバーヘッドにより性能が落ちる説について

今日の話

- データプレーンのソフトウェア技術：ソフトウェアルータ, NFV
  ≠コントロールプレーン: SDN, White box switch
  - ソフトウェアによる高速パケット処理技術
    ～ハードウェアアーキテクチャを知る～
  - NFVに向けて
    ～仮想化を知る～

ソフトウェアによるパケット処理

- そもそも「ソフトウェア（英: Software）」とは？
  - Software: “noun: *the programs and other operating information used by a computer.*” (quoted from New Oxford American Dictionary)
  - Hardware: “noun: *the machines, wiring, and other physical components of a computer or other electronic system.*” (quoted from New Oxford American Dictionary)

パケットフォワーディングの基本処理（プログラム）

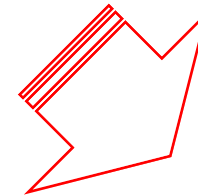
- Ethernet, IPにおけるフォワーディング
  - Store-and-Forward (≠ Cut-Through) ＝**パケットの「コピー」**

- 転送先の決定
  - Ethernet: Forwarding Database (FDB) / MAC Address Tableの参照・更新
  - IP: ARP TableとRouting Tableの参照・更新
    - L3スイッチでは高速化のため、これらのテーブルをForwarding Information Base (FIB)に統合

- パケット処理
  - ヘッダ処理: TTL減算, 宛先MACアドレス書き換え
  - カプセル化: VLAN, VxLAN, etc
  - 暗号化: IPsec
  - フラグメント: IP fragmentation

パケットフォワーディングの基本処理（プログラム）

- Ethernet, IPにおけるフォワーディング
  - Store-and-Forward (≠ Cut-Through) ＝**パケットの「コピー」**
- 転送先の決定
  - Ethernet: Forwarding Database (FDB) / MAC Address Tableの参照・更新
  - IP: ARP TableとRouting Tableの参照・更新
    - L3スイッチでは高速化のため、これらのテーブルをForwarding Information Base (FIB)に統合
- パケット処理
  - ヘッダ処理: TTL減算, 宛先MAC アドレス書き換え
  - カプセル化: VLAN, VxLAN, etc
  - 暗号化: IPsec
  - フラグメント: IP fragmentation



計算コストの高い処理は（ほぼ）ない

- 計算コストの高い処理：乗算・除算, 浮動小数点演算，行列演算，微分積分 etc…
- IPsec (AES) はハードウェアオフロードあり

パケットフォワーディングプログラムの何/何処が大変なのか？

- パケットフォワーディングプログラムの特徴
  - 高レート処理 = CPU-intensive & Latency-bound programming
    - I/O-intensiveではない
      - 高々数100 Gbps; 100 Gbps = 12.5 GB/s
        - DDR4-2133: 17 GB/s per channel
      - （後述の）メモリコピーがパフォーマンスボトルネックとなるのはI/O帯域ではなく、メモリコピー命令のLatencyが大きいため

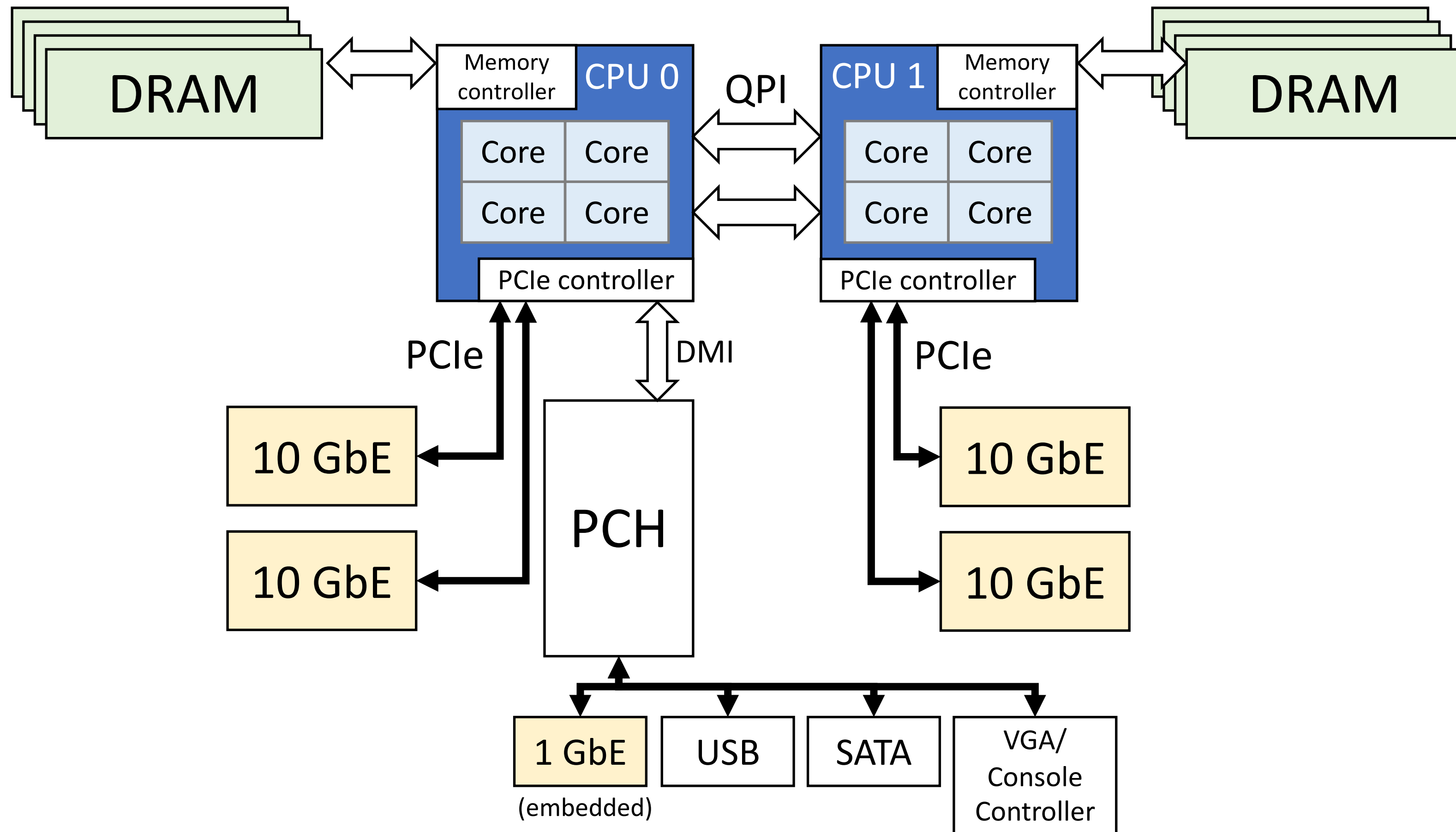
Ethernetにおけるパケットレート(Mpps) (Inter-frame gap = 12 Bytes)

	64-Byte	1518-Byte
1 GbE	1.488	0.08127
10 GbE	14.88	0.8127
25 GbE	37.20	2.032
40 GbE	59.52	3.251
100 GbE	148.8	8.127

パケットフォワーディングのデータパス

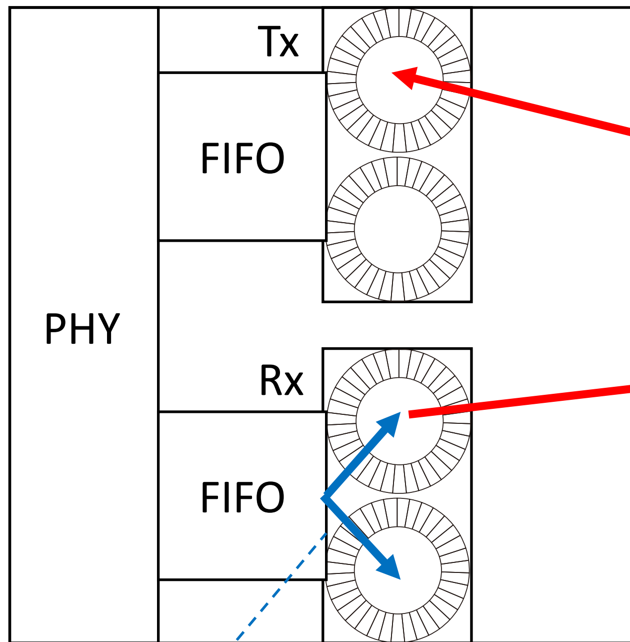
汎用機のアーキテクチャに沿って説明します

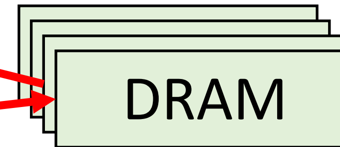
汎用機のアーキテクチャ



(Intel® Broadwell Generation's Architecture; e.g., Xeon v4 series)

汎用NICのアーキテクチャ



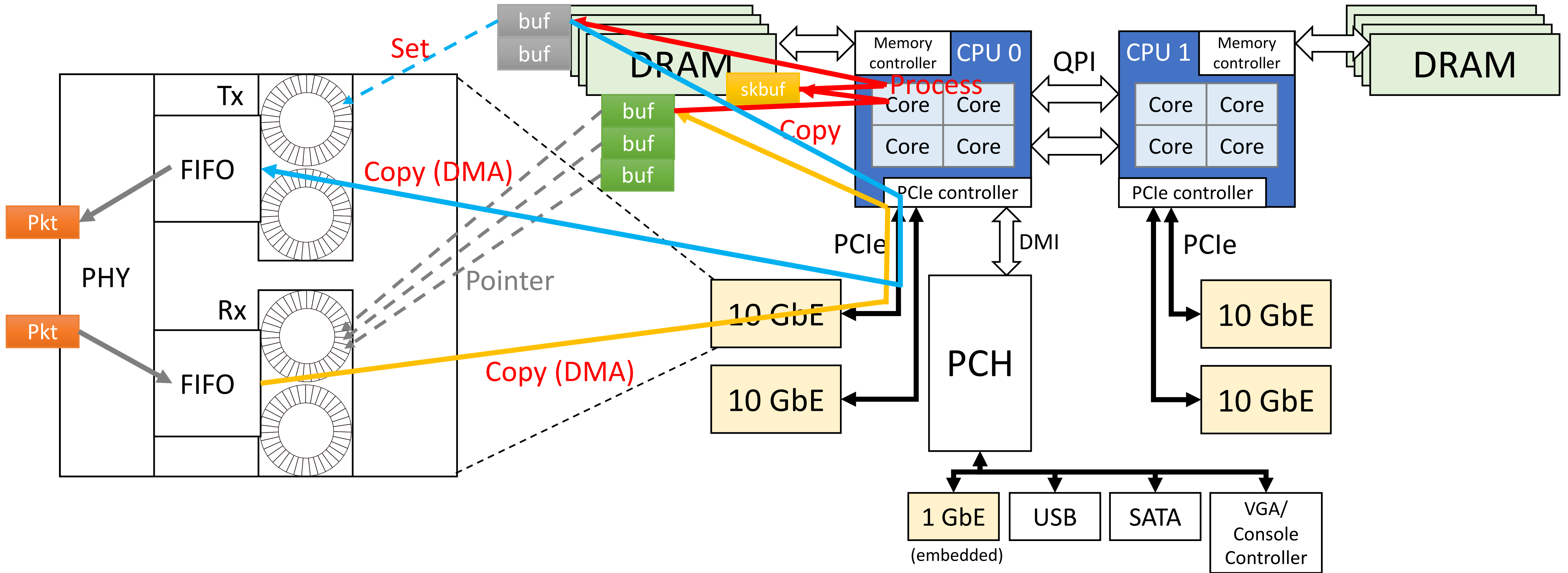


- Txバッファにパケットへのポインタをセット
- パケットの送信が完了すると
  - CPUに割込みを送る

- RxバッファにDMA先のポインタを予めセット
- パケットが到着すると
  - DMAでDRAMに書き込む
  - CPUに割込みを送る

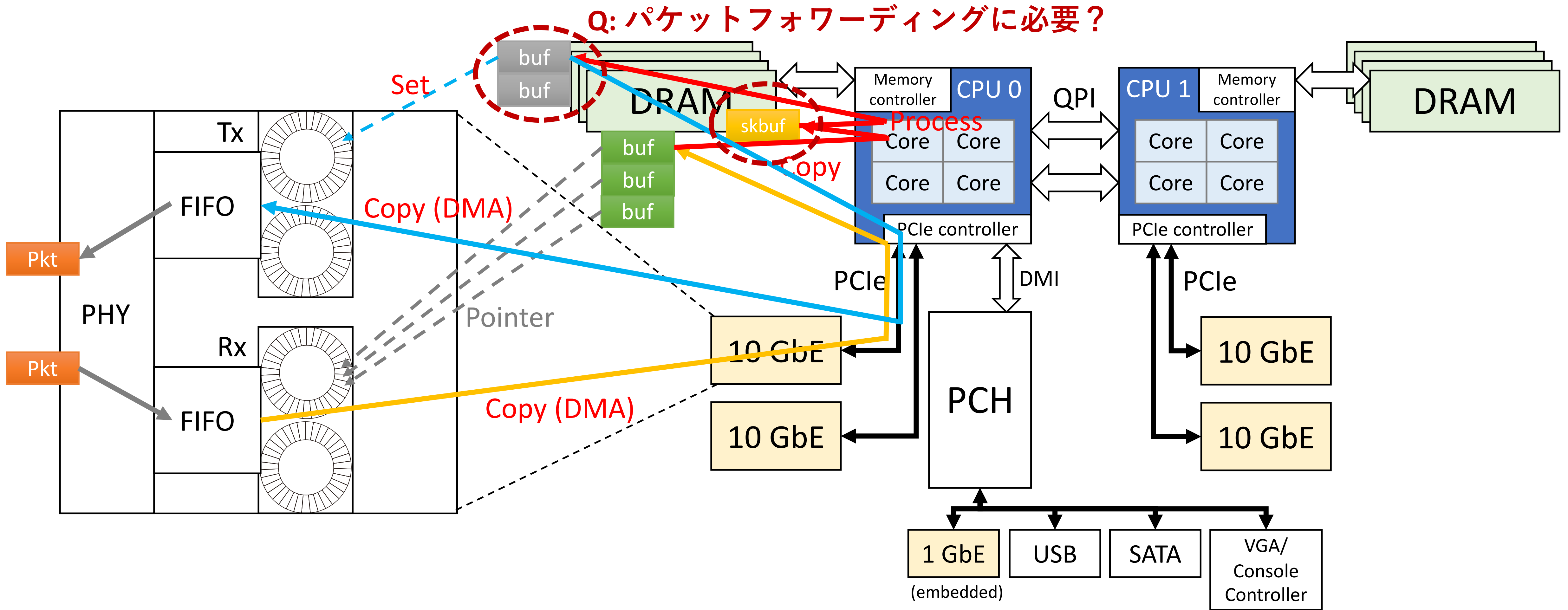
1 NICにつき複数の送信(Tx)・受信(Rx)バッファ
- Intel X520シリーズ: 128 Tx queues, 1536 Rx queues
- Intel XL710シリーズ: 768 Tx-Rx queue pairs

データパス（ハードウェア視点）



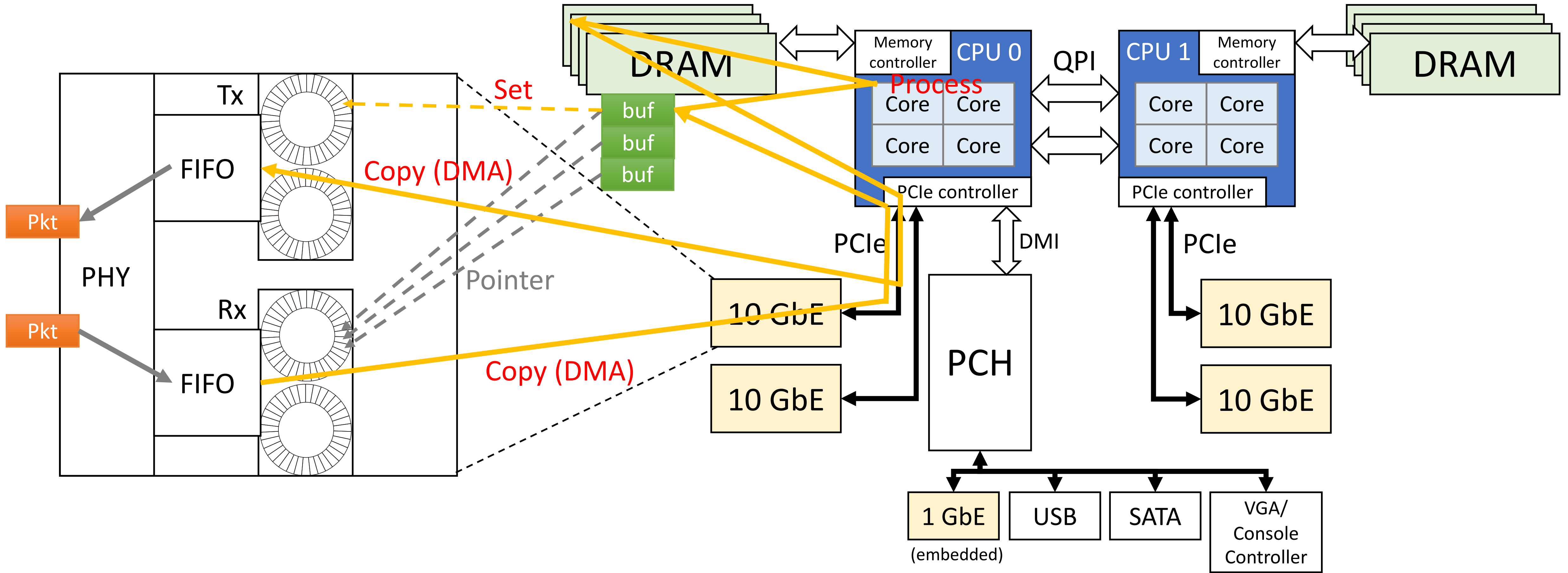
(Intel® Broadwell Generation's Architecture; e.g., Xeon v4 series)

データパス（ハードウェア視点）



(Intel® Broadwell Generation's Architecture; e.g., Xeon v4 series)

Zero-copyデータパス (DPDK/netmap)



(Intel® Broadwell Generation's Architecture; e.g., Xeon v4 series)

割込みの四方山話

割込み

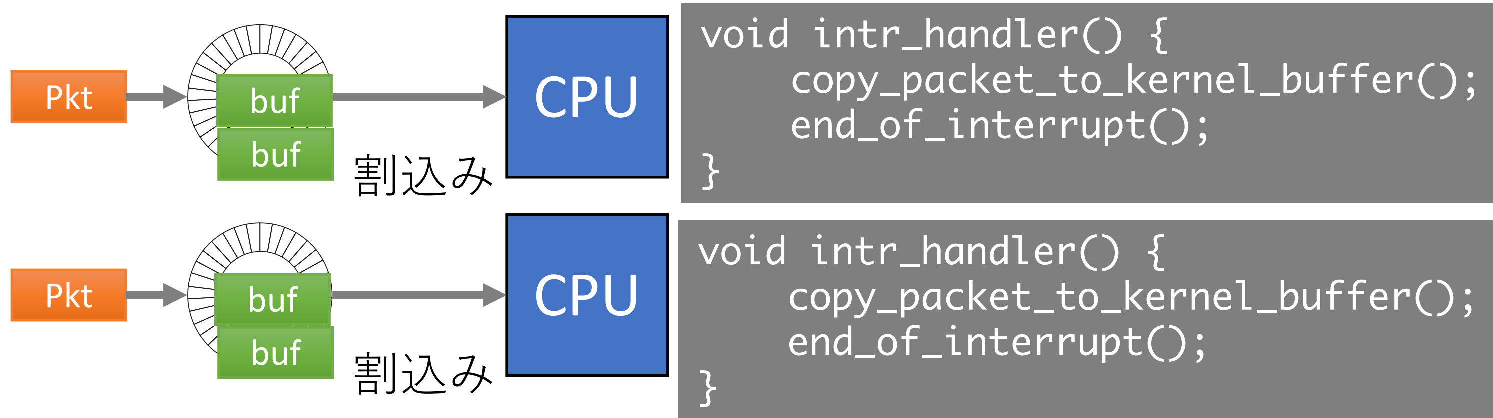
- Preemptive multitasking OS
    - *BSD, Linux, macOS, etc.
  - 一定時間ごとにOSがタスクを切り替えるマルチタスク実装
  - 割込みによるタスクスイッチ
    - Preemption timerとしてタイマー割り込みを使う
      - 100–1000 Hzが一般的

  - 外部割込み
    - 外部入力装置（キーボードやシリアルコンソール）などの入力を受け付けたら
      - デバイス固有の処理を行う→割込みハンドラ
      - デバイスを管理するドライバタスクを起こす

外部割込み

- そもそもキーボードなど（速くても100 Hz程度）を想定
  → かなり高コスト
  1. 現在のタスクのコンテキストを保存（>10命令）
  2. 割込みハンドラの呼び出し
     - デバイスからのdata load/storeなど
     - 割込み処理の完了をデバイスに通知（PCIeの場合100マイクロ秒程度）
  3. 割込みの完了を割込みコントローラに通知（ナノ秒オーダー）
  4. 保存したコンテキストを戻す（>10命令）
  5. タスクの再起動

割込み処理高速化の歴史：～100BASE-TX

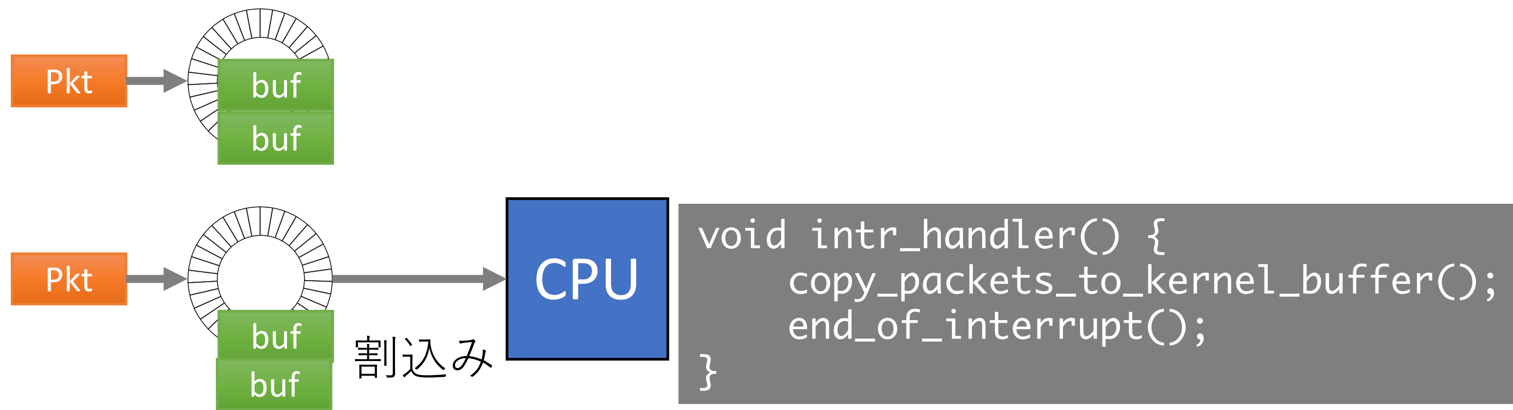


- 1パケットの送受信ごとに割込み
  - 割込みハンドラ内でパケットの処理を行う

(参考) Ethernetにおけるパケットレート(Mpps)
(Inter-frame gap=12-Byteとする)

	64-Byte	1518-Byte
1 GbE	1.488	0.08127
10 GbE	14.88	0.8127
25 GbE	37.20	2.032
40 GbE	59.52	3.251
100 GbE	148.8	8.127

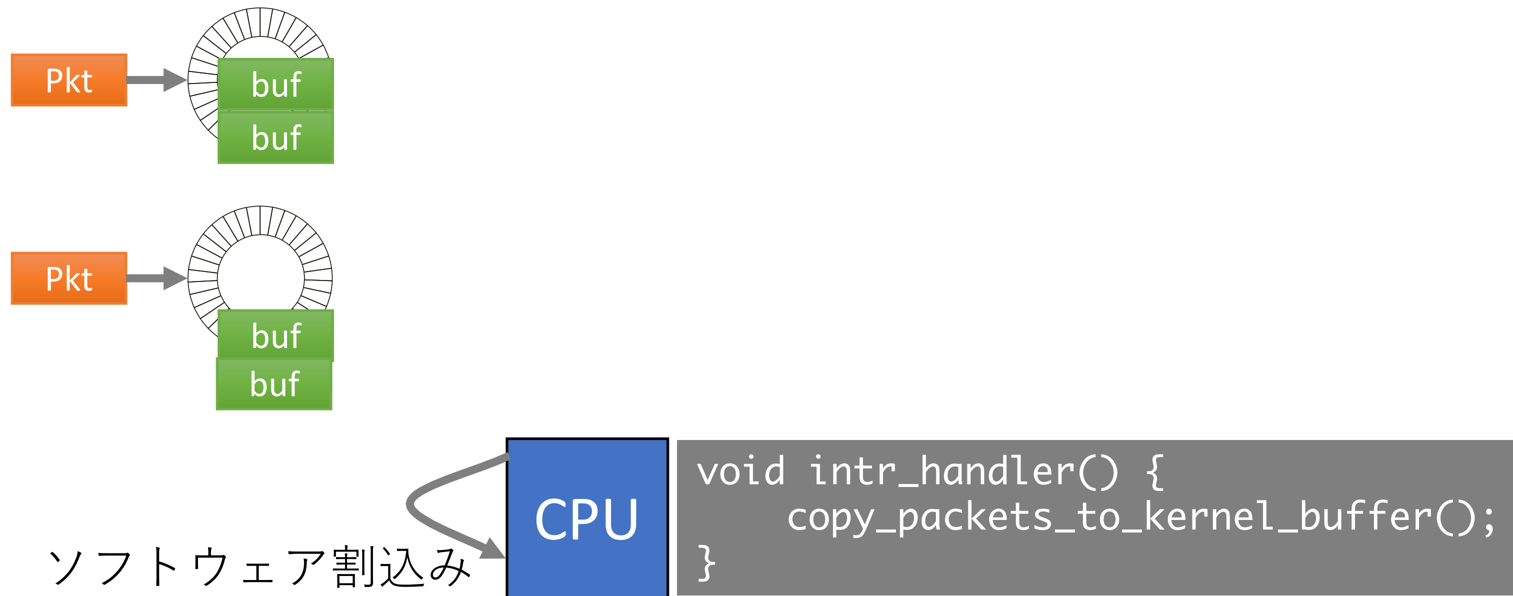
割込み処理高速化の歴史：Interrupt Throttling



- 連続する数パケットの送受信ごとに割込み：Interrupt Throttling
  - 一定時間以内の割込みは抑制
  - 割込みハンドラ内で受信済みパケットの処理を行う

(参考) Inter-interrupt interval: 2–1024 μs Intel® X520 10 GbE NIC
→ ≤1–500 kHz

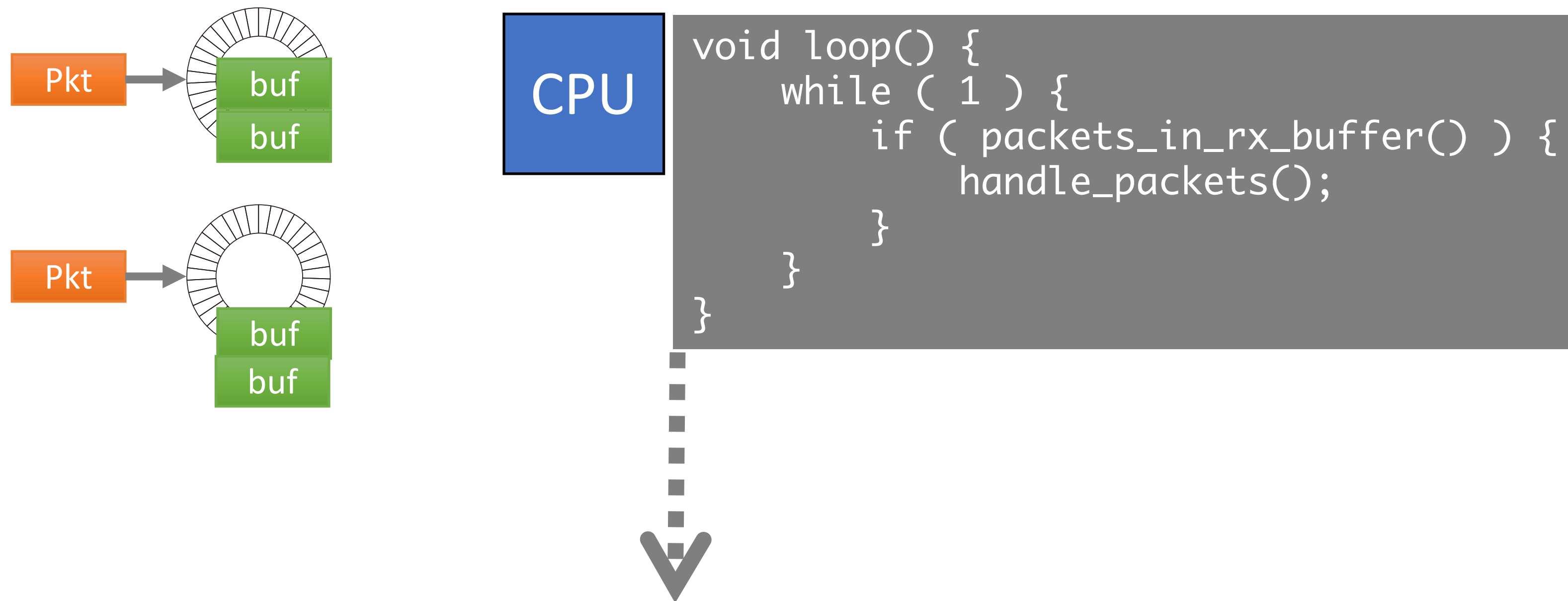
割込み処理高速化の歴史：Linux NAPI




```
void intr_handler() {
    copy_packets_to_kernel_buffer();
}
```

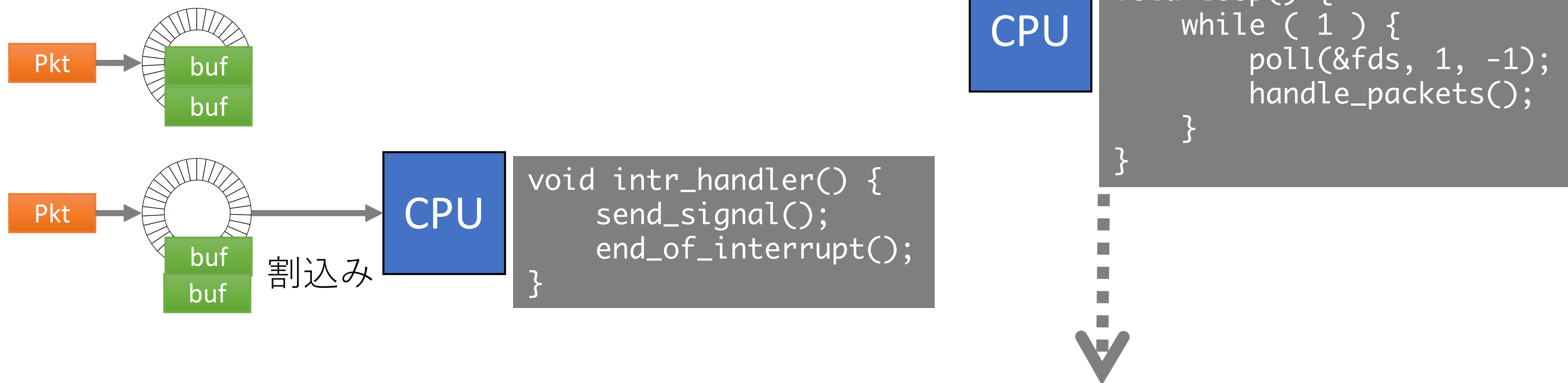
- 割込みを無効化し、タイマーで割込む（ソフトウェア割込み）＝**ポーリング**
  - （ソフトウェア割込みの）割込みハンドラ内で受信済みパケットの処理を行う

割込み処理高速化の歴史：DPDK



- 割込みを無効化し、ポーリングをし続ける＝**ビジーウェイト**
  - ポーリングをし続けるタスクが受信済みパケットの処理を行う

割込み処理高速化の歴史：netmap



- 連続する数パケットの送受信ごとに割込み
  - 一定時間以内の割込みは抑制
  - 割込みハンドラ内ではPOLLINシグナルを生成する処理だけを行い、パケット自体の処理は行わない
  - netmapデバイスを開いているユーザランドプログラムが受信済みパケットの処理を行う（poll()システムコール）

各技術の割込み処理方法のまとめ

	割込み頻度	パケット処理タスク	コピー	性能
Original	1パケットごと	割込みハンドラ	あり	×
Interrupt Throttling	一定時間に1(or 0)回	割込みハンドラ	あり	×
Linux NAPI （ポーリング）	ソフトウェア割込み	カーネルタスク	あり	△
DPDK（ビジーウェイト）	なし	ユーザランド	なし	○
netmap（select()/poll()）	一定時間に1(or 0)回	ユーザランド	なし	○

- 割込みが重いは「本当」
  - MHz単位は扱えない
  - kHz単位ならOK (e.g., netmap)
- メモリのコピーの方が性能に大きく寄与
  - カーネルバッファへのコピー
    → パケットのまま扱う（厳密にはring bufferのdescriptor単位）

高性能プログラミングに関係する用語

- スーパースカラ型プロセッサ
  - パイプライン：命令をデコードや実行、メモリアクセスなど複数のステージに分割しスループットを向上する技術
  - アウトオブオーダー実行・完了：コードを複数の実行器により並列実行するときに（依存関係（ハザード）の無い範囲で）実行・完了できるものから順に実行することで、待ち時間を削減する技術（↔インオーダー実行）
- CPUキャッシュ
  - L1/L2/L3(Last-Level) Cache
    - L1キャッシュ：低レイテンシ・小容量
    - LLC：高レイテンシ・大容量
  - セットアソシアティブ型キャッシュ
  - アソシアティビティ（Associativity）
  - True/False Sharing
- MMU (Memory Management Unit) / TLB (Translation Lookaside Buffer)
  - Page Table: 仮想アドレスから物理アドレスへの変換を行うテーブル
  - TLB: Page Tableのキャッシュ（非常に小さい; e.g., 64エントリ）

ソフトウェアによる高速パケット処理プログラミングガイドライン

- メモリコピーは最小限で
  - メモリコピー：キャッシュ汚染+高レイテンシ
- バッチ処理
  - FDBやFIB lookupアルゴリズムのキャッシュ効率の向上
  - パイプライン効率化: NICへのI/O命令はインオーダー実行のために同期が必要（=高コスト）なため、NICへのI/O命令の回数を減らしたい(I/O命令：netmapでのrx_sync()/tx_sync()など)
- 並列プログラミング
  - ロックフリーを心がける
  - True/False sharingを回避（True sharingはやむを得ない場合もありますが……）
- （DPDKやnetmapに頼らない人は）
  - CPUのキャッシュアソシアティビティを考慮したデータ配置
  - リングバッファ情報の書き戻しタイミングの制御によるPCIe資源の節約
  - Huge table（2 Mbyte/1 Gbyte-paging）によるTLBミスの削減
  - 割込みハンドラの最適化

ソフトウェアによる高速パケット処理プログラミングガイドライン

（あくまでもOS実装時の経験則ですが）
CPUや構成を変えて性能が出ない場合
CPUキャッシュが影響していることがほとんどした

例）ポート数を増やしたら性能出ない
　→原因）パケットバッファが増えてキャッシュが足りない（汚染される）
例）複数のTx/Rxバッファを使ったら性能/coreが下がった
　→原因）ポートごとに保持していたリングバッファの構造体（配列）で
　　False sharingが起こっていた

今日の話

- データプレーンのソフトウェア技術：ソフトウェアルータ, NFV
  ≠コントロールプレーン: SDN, White box switch

  - ソフトウェアによる高速パケット処理技術
    ～ハードウェアアーキテクチャを知る～

  - NFVに向けて
    ～仮想化を知る～

仮想化とは？

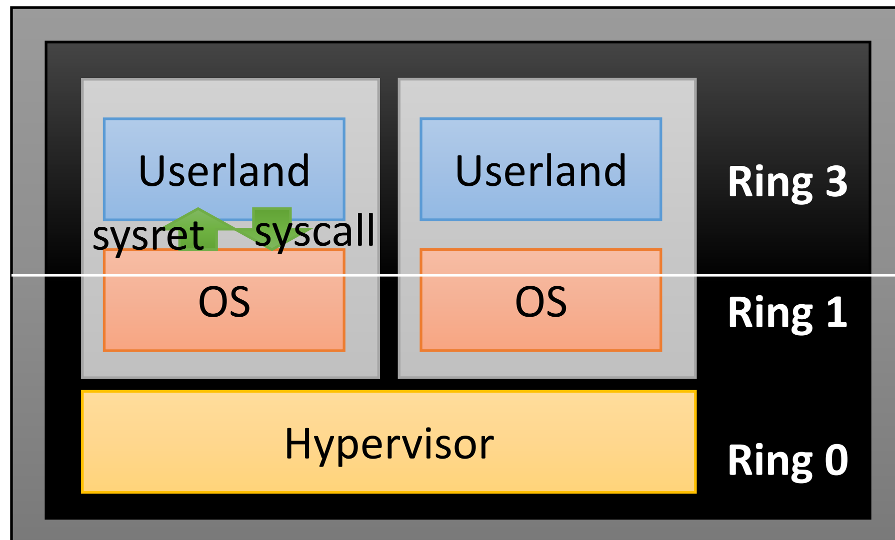
- ソフトウェアにより「仮想的に」ハードウェアの機能を実現する
    - 仮想CPU
    - 仮想メモリ
    - 仮想NIC
    - 仮想ストレージ
    - 仮想タイマ
    - etc…

ハードウェアによる仮想化（支援）

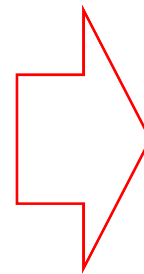
- ハードウェアによる仮想化支援技術登場前の仮想化
  - CPU・メモリなど全てをソフトウェアエミュレーション（e.g., QEMU）
→ 低性能
  - 準仮想化（e.g., Xen）：CPU・メモリはそのまま使う
→ （CPU・メモリ資源保護のため）
仮想マシン上で動かす対象のOSへの変更が必要

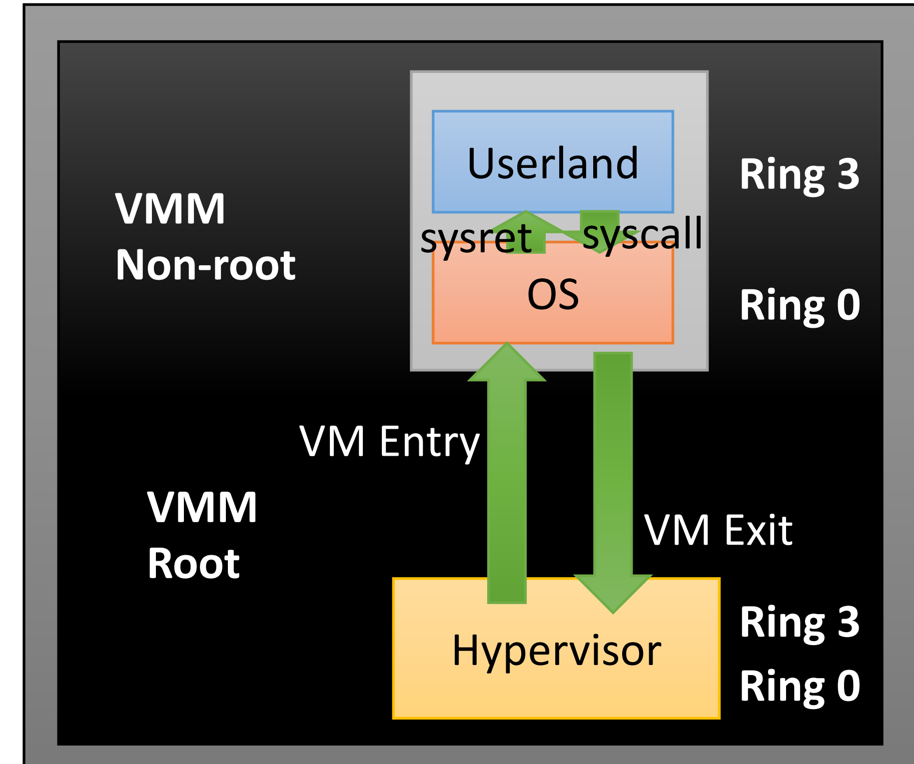
ハードウェアによる仮想化（支援）

CPU・メモリについての仮想化支援：Intel VT-x / AMD-V
- CPUはVMM Root/Non-rootにより実現
- メモリはExtended Page Tableにより実現
  仮想メモリの物理アドレス→物理アドレス



Generic design of hypervisor with IA Ring protection architecture without hardware's virtualization assist





Generic design of hypervisor with hardware's virtualization assist (Intel® VT-x)

ハードウェアによる仮想化（支援）

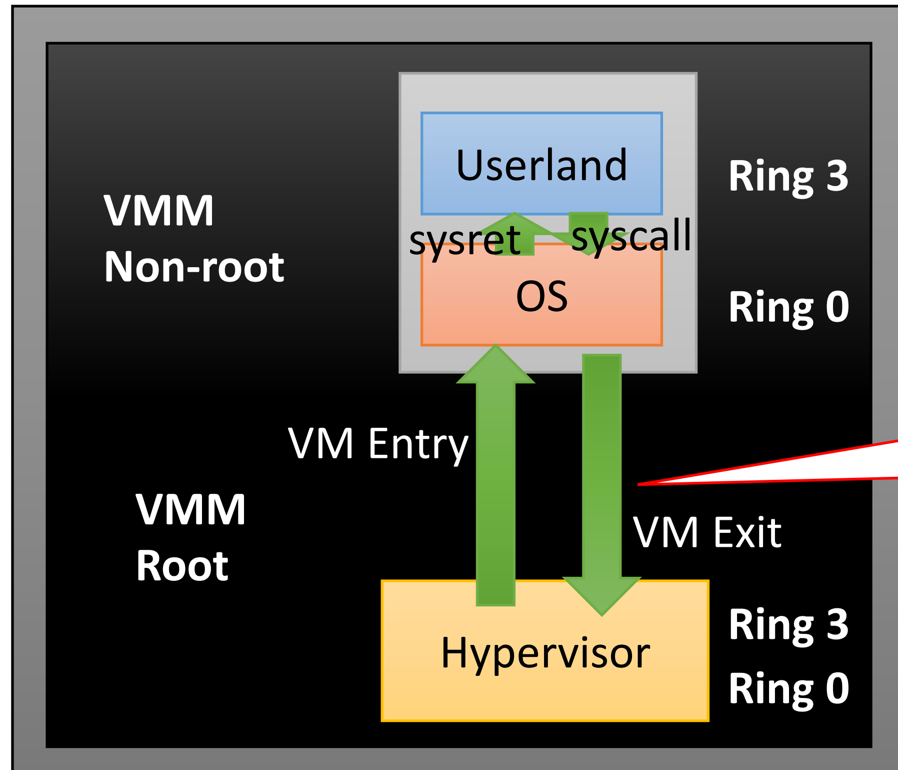
- NICの仮想化
  - IOMMU (Intel VT-d / AMD-VI)
    - CPUの機能
    - PCIデバイスを特定のVMから操作できるようにする仮想化支援技術（PCI passthrough）
  - SR-IOV
    - NICの機能
    - 物理NIC（Physical Function）を複数の仮想NIC（Virtual Function）として扱えるようにする仮想化技術
  - Virtual Switch
    - NICの機能
    - 仮想NIC間および仮想NICと外部ポート間のEthernetスイッチ機能を提供

仮想化によるオーバーヘッド？

- 仮想化=ソフトウェアにより「仮想的に」ハードウェアの機能を実現する
  - 仮想CPU → Intel VT-x / AMD-V
  - 仮想メモリ → Intel VT-x / AMD-V
  - 仮想NIC → IOMMU + SR-IOV + Virtual Switch
  - 仮想ストレージ → 今回は対象外とします
  - 仮想タイマ → Intel VT-x / AMD-V
  - etc…

ネットワーク機能仮想化に必要なものはハードウェア支援により
仮想マシンから透過的に（ソフトウェアエミュレーションなしに）
ハードウェアを扱うことができる
→ 仮想化によるオーバーヘッドとは何か？

仮想化のオーバーヘッド



- VM-Exit/VM-Entry：
  仮想マシンレベルでのコンテキストスイッチ
- VMCS (Virtual-Machine Control Structure)：
  GDT、IDT、コントロールレジスタなど
  VMのコンテキスト情報の一部を保存するデータ構造
  (4-KiB)

1. VMCSの保存
2. 汎用レジスタの保存
3. FPU/AVXレジスタの保存（必要に応じて）
4. VM-Exitの処理（例：ハイパーバイザコールの実行）

Note: VM-Exit以外にEPTのTLBミスなども
（VM-Exitと比較して軽微ではあるが）
仮想化のオーバーヘッドとなる

VM-Exitのオーバーヘッド

- Intel Core i7 6700Kでの実測値
  - 実験1
    - VMではHLT命令のループ
    - VMMではVMCS、汎用64ビットレジスタの保存・リストア

    → 1795 tsc/vm-entry-exit-set ≒ 0.45 μs/vm-entry-exit-set

  - 実験2
    - 実験1をnested VM上で実行（VMMはUbuntu 14.04上のKVMを利用）

    → 28383 tsc/vm-entry-exit-set ≒ 7.1 μs/vm-entry-exit-set

主なVM-Exitの発生条件

- External Interrupt
  - 抑制可能だが一般的なVMMはVM-Exitを発生
- Triple fault
- INIT/Start-up IPI/System-management interrupt
- CPUID/GETSEC/HLT/INVD/INVLPG/RDPMC/RDTSC/RSM/RDRAND/VMxxx命令など
  - RDTSC/RDRANDなどは抑制可能
  - HLTも抑制可能だが一般的なVMMはVM-Exitを発生
- Control Register操作
- Debug Register操作
- I/O命令
- RDMSR/WRMSR命令
- APIC操作
- preemptionタイマーのタイムアウト

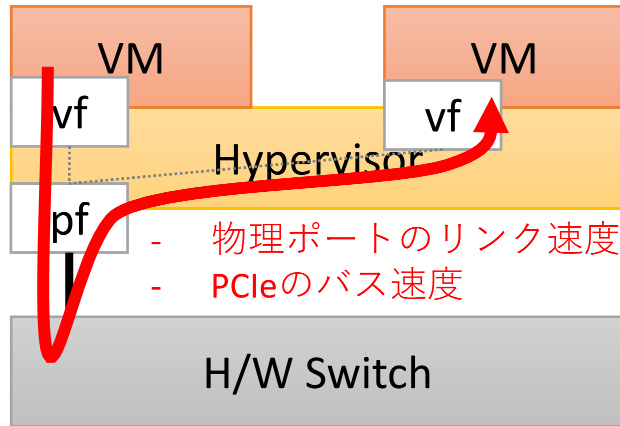
VNFの高速化テクニック

- ハードウェア仮想化支援を活用する
  - SR-IOV + IOMMU (PCI passthrough)

- VM-Exitの頻度を減らす
  - タイマ割込み頻度（Hz）を減らす / Ticklessにする
  - VM-Exitを発生させる命令・システムコールに注意する

VM-to-VMの通信の最適化

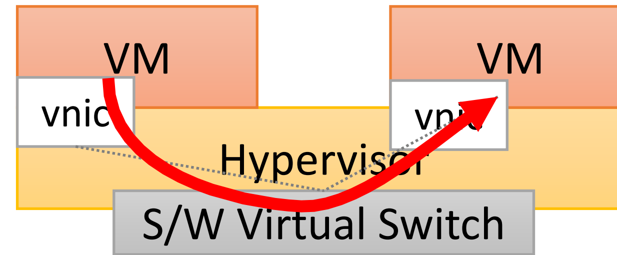
1a) ハードウェアスイッチを介した通信
（SR-IOV+IOMMU）



- 物理ポートのリンク速度
- PCIeのバス速度

➔ **高ppsなVNFに最適**

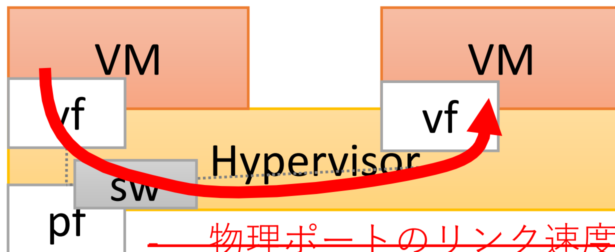
2) ソフトウェアスイッチを介した通信
（ソフトウェア仮想スイッチ）



- 「共有」メモリへのコピー（w/ TCP Segmentation Offload）
- VM-Exitによるハイパーバイザへのスイッチング処理

➔ **低pps高bpsなTCPを使ったVNFに最適**

1b) NIC内蔵のハードウェアスイッチを介した通信



- ~~物理ポートのリンク速度~~
- PCIeのバス速度

➔ **高ppsなVNFに最適**

まとめ

- ベアメタルにおけるソフトウェアによるパケット処理の高速化
  - メモリコピーを減らす
  - キャッシュ効率を考慮する
  - NICの外部割込みは無効にするかInterrupt Throttlingを使う
- 仮想環境におけるソフトウェアによるパケット処理の高速化
  - ハードウェア仮想化機能を活用する
  - VM間の通信はパケットレート重視ならホスト内のソフトウェアスイッチよりも外部のハードウェアスイッチを介した方が良い
  - VM-Exitに注意する