Mitosis

Transparent Self-Replicating Page Tables

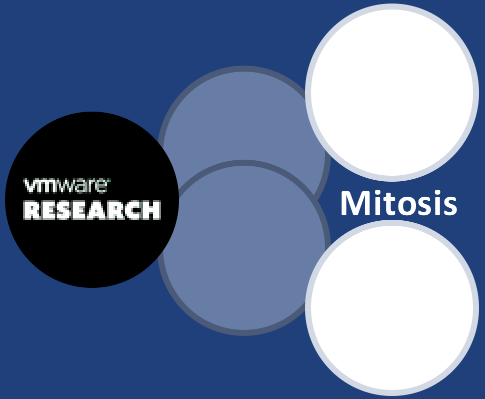
Reto Achermann

VMware Research Group

Summer 2018

Mitosis - Transparent Self-Replicating Page Tables

Reto Achermann[1,2], Jayneel Gandhi[1], Timothy Roscoe[2], Abhishek Bhattacharjee[3]
[1]VMware Research Group, [2]ETH Zurich, [3]Rutgers University



TL;DR

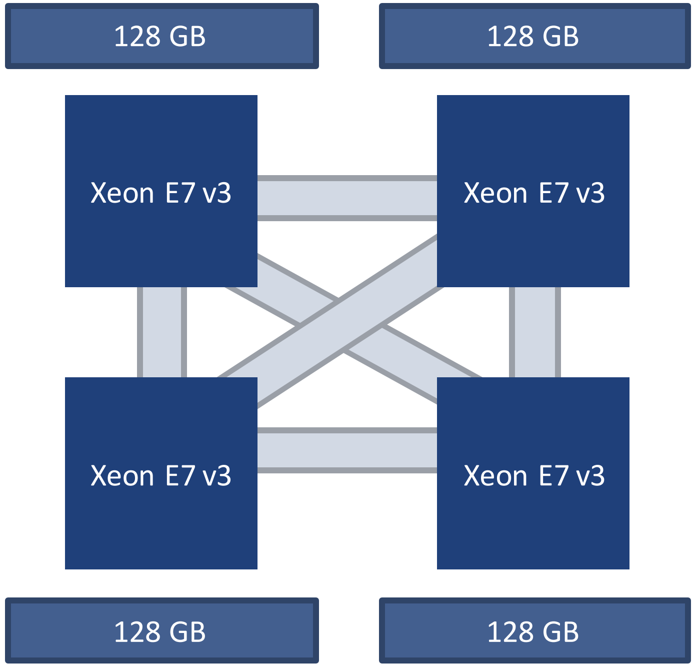
- Page table walks can account for a large fraction of the runtime
- Non-uniform memory access (NUMA) effects increase page walk time

- **Mitosis: reduce NUMA effects on page table walks through page table replication**

- Up to 3.4x improvement over worst case scenario
- Up to 15% improvement on multi-threaded workloads

- Without program modifications

Target Hardware Configuration: Big Memory Machines



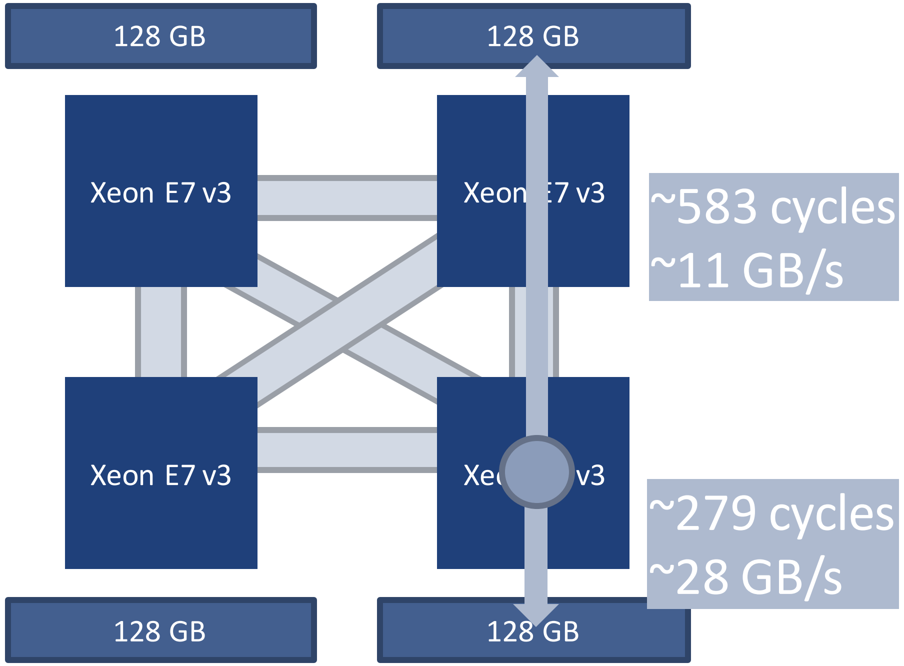
Host: vrg-10 / vrg-11
4x14x2 CPU E7-4850 v3 @ 2.20GHz
512 GB RAM

Bandwidth & capacity limited per processor socket

- More bandwidth & capacity ⇔ Multi-socket machines
- Up to 16 sockets possible max. 24TB RAM

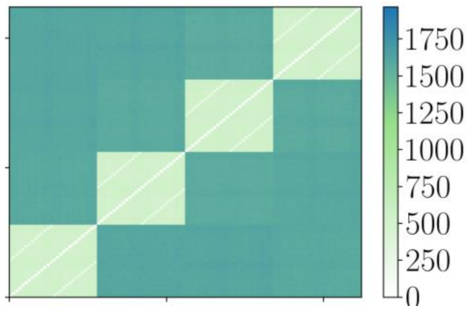
Big Memory Machine Characteristics

About half the bandwidth and double the latency to remote node



Host: vrg-10 / vrg-11
4x14x2 CPU E7-4850 v3 @ 2.20GHz
512 GB RAM

Smelt (OSDI’16)



16-socket machines this can be 1000 cycles latency

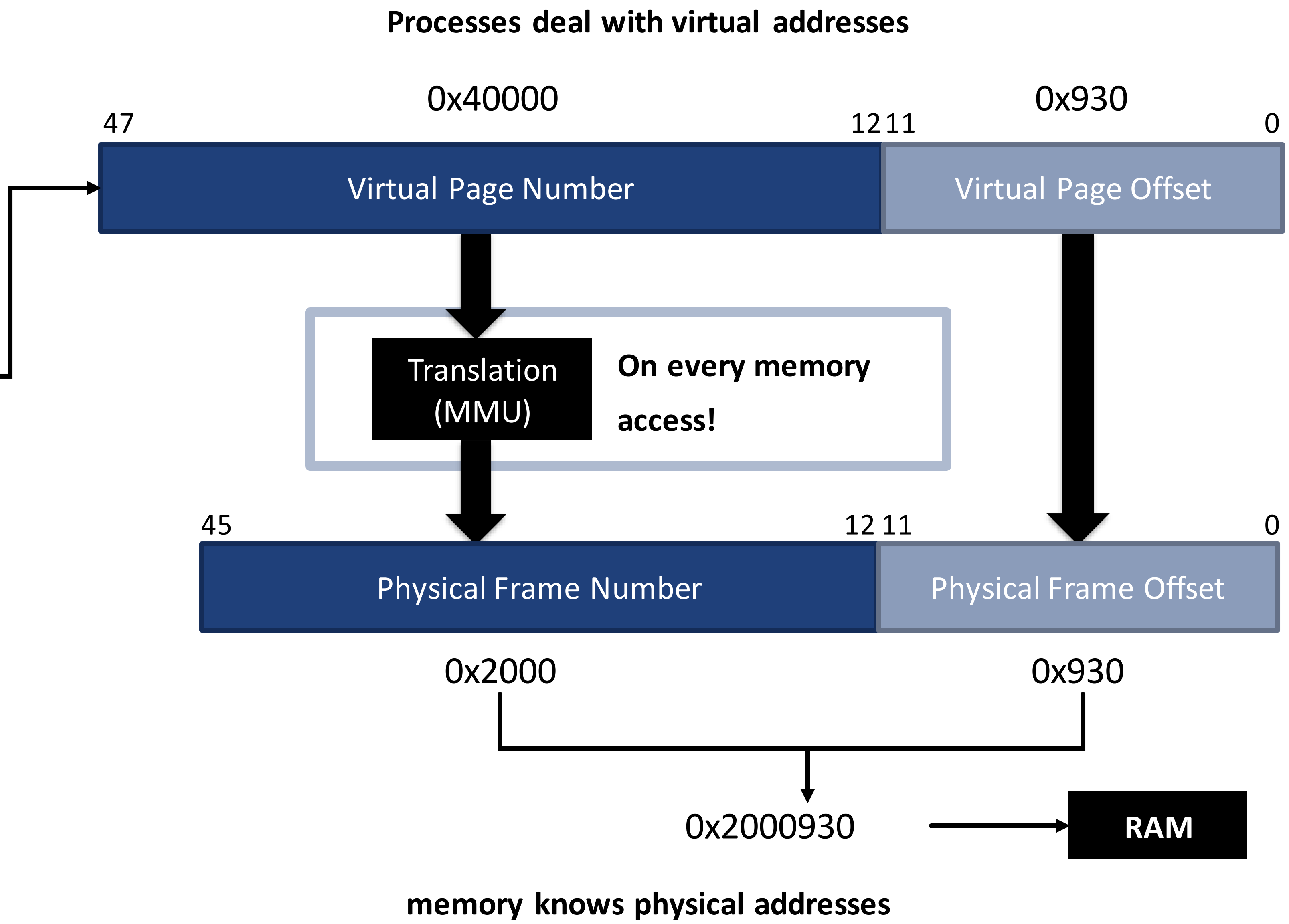
Data Allocation Strategies for NUMA Machines

- Well studied in literature
  - Carrefour (Dashti et al)
  - Blackbox Concurrent Data Structures (Calciu et al)
  - Shoal (Kaestle et al)
- Different policies in the OS (numactl)
  - First touch (local allocation)
  - Interleave
  - mbind

Focus mainly on data allocation, ignore page table placement

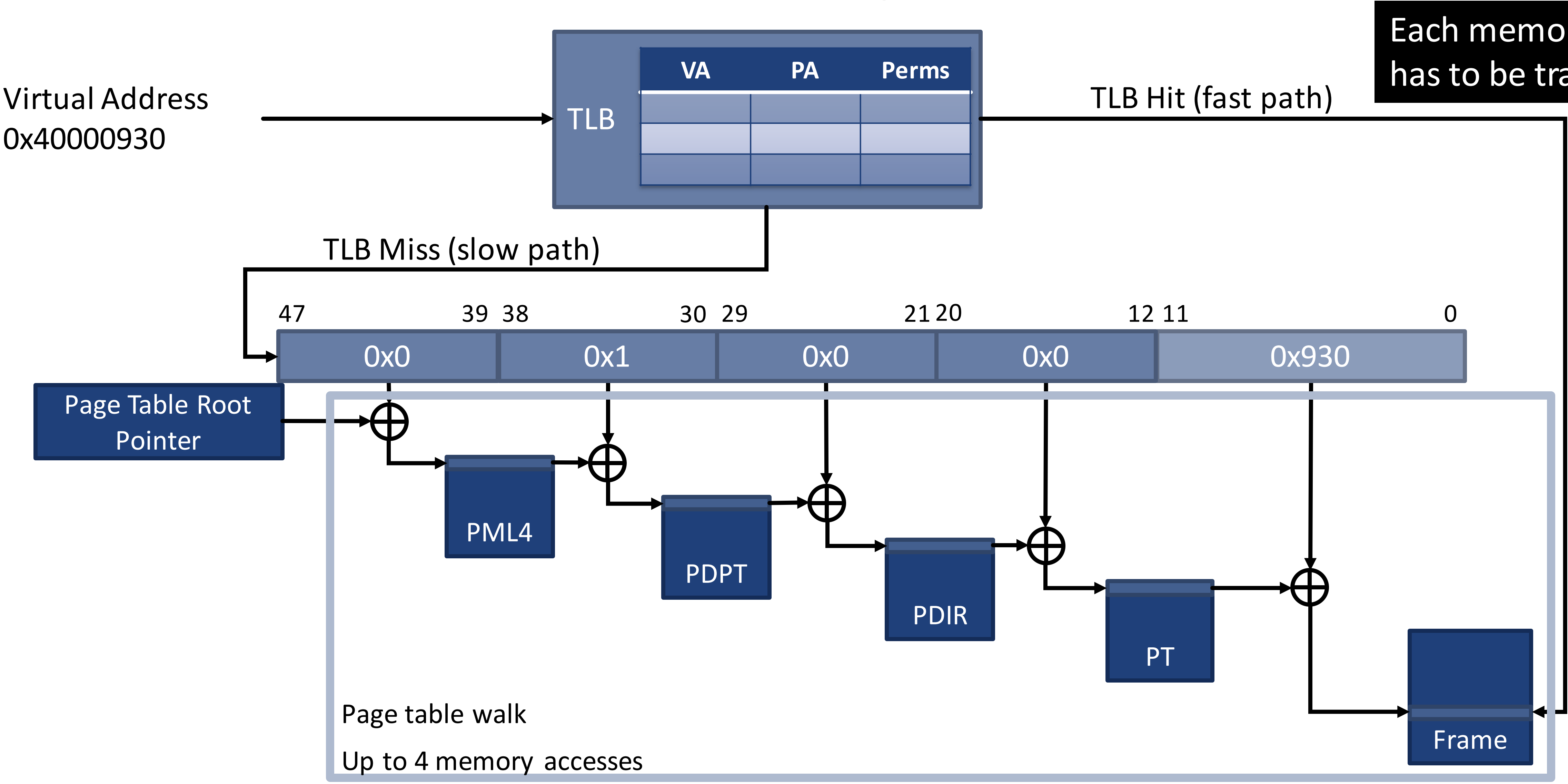
Virtual to Physical Address Translation

```
int my_awesome_function(void)
{
    uint64_t *foo = malloc(4096);
    // foo = 0x40000930

    foo[294] = 42;

    // ...
    return 0;
}
```



Translation Lookaside Buffers – Caching Translations since 1965

Each memory access has to be translated!



TLB Reach is Limited

Lookup for every memory access!

Fast cache to store the resolved translations. Overlaps L1 cache access.

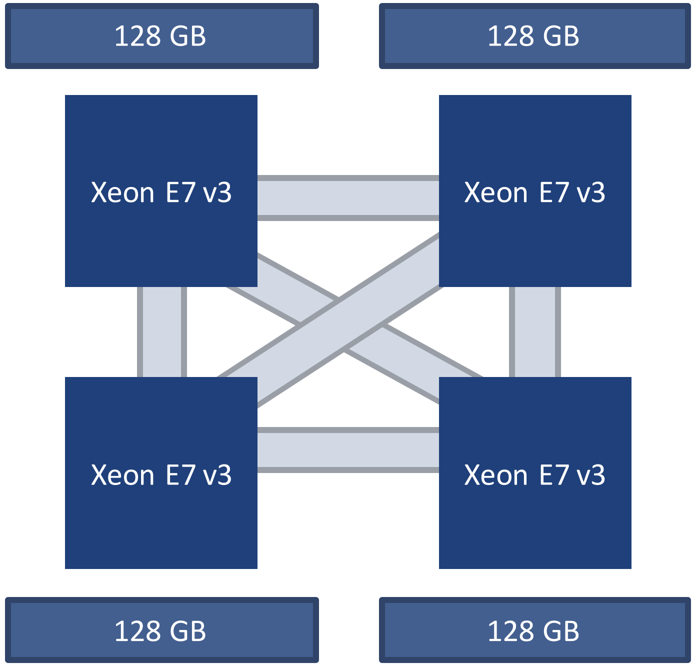
It’s **tiny** ! RAM capacity is growing faster than TLB capacity

On our machines:

TLB reach:	(64+1024)4K / 512G = 0.008%	(4k pages)
	(32+1024)2M / 512G = 0.4%	(2M huge pages)
	(4+0)1G / 512G = 0.78%	(1G huge pages)

How does NUMA affect the page walk time?

Micro Benchmark: Effects of Page Table Placements



Host:
4x14x2 CPU E7-4850 v3 @ 2.2GHz
512GB RAM

OS: Modified Linux kernel

- Force page table allocation to node 0

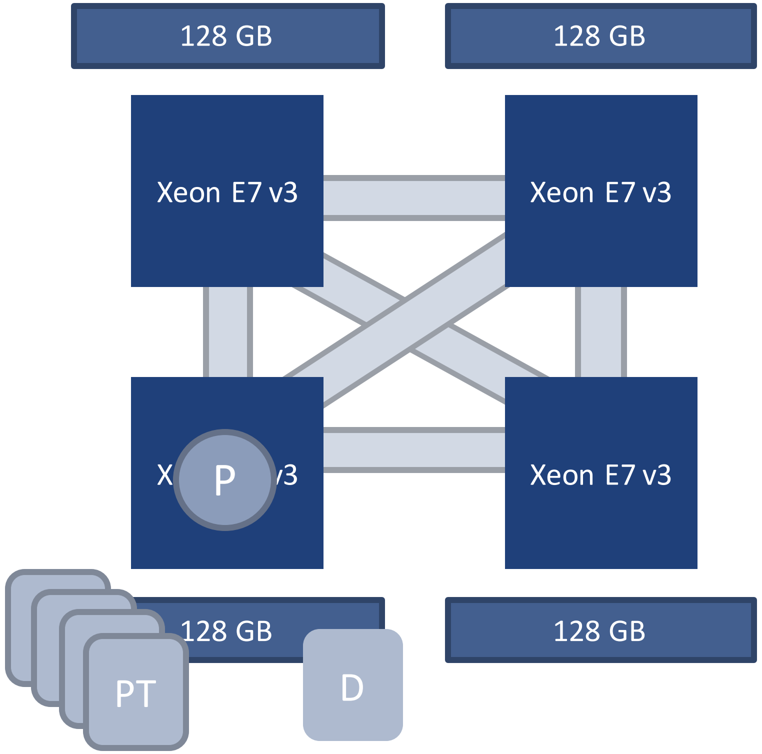
Runtime: use numactl / libnuma to

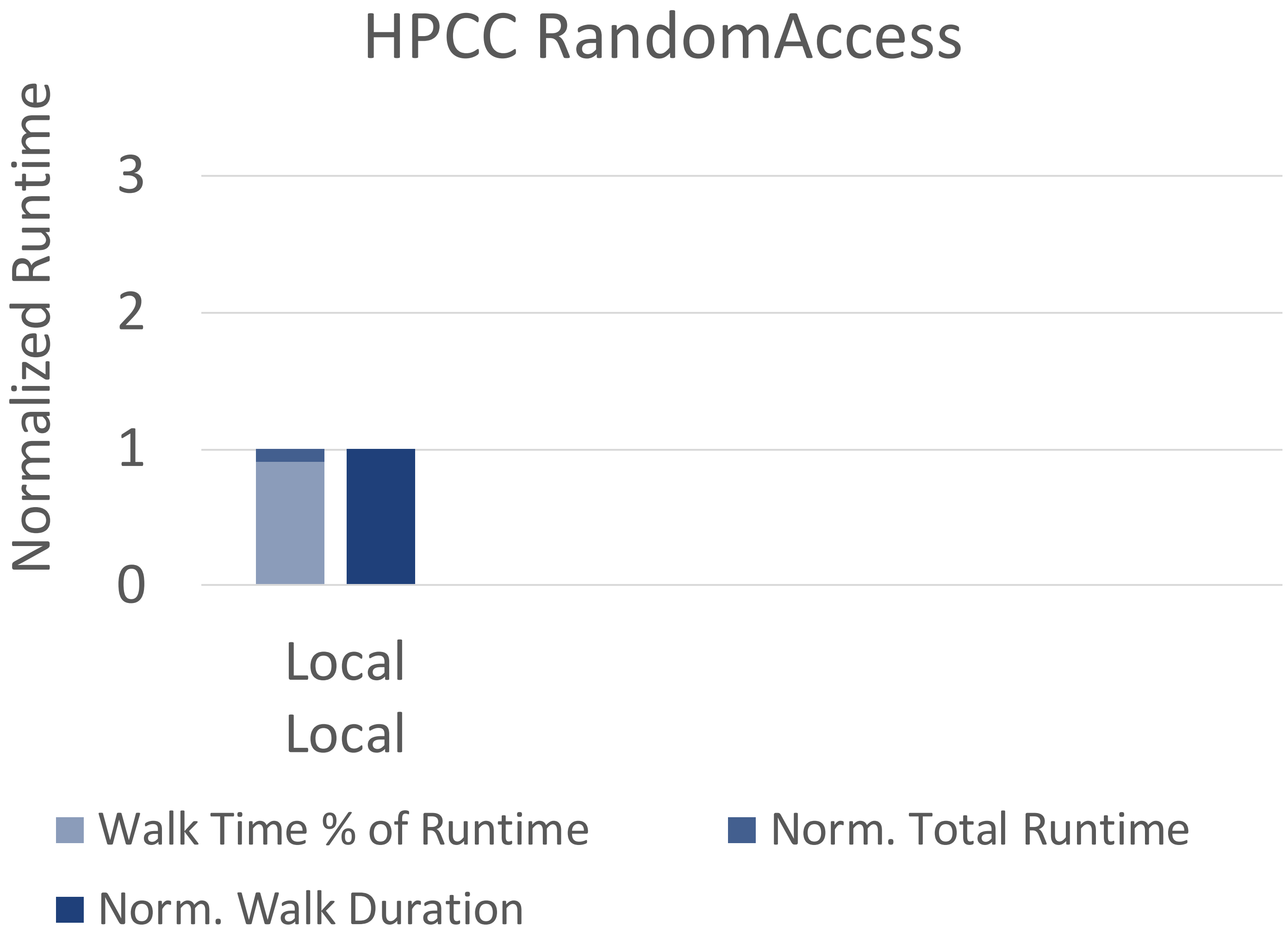
1. Restrict where the program runs
2. Allocate data from a fixed node

Workload:

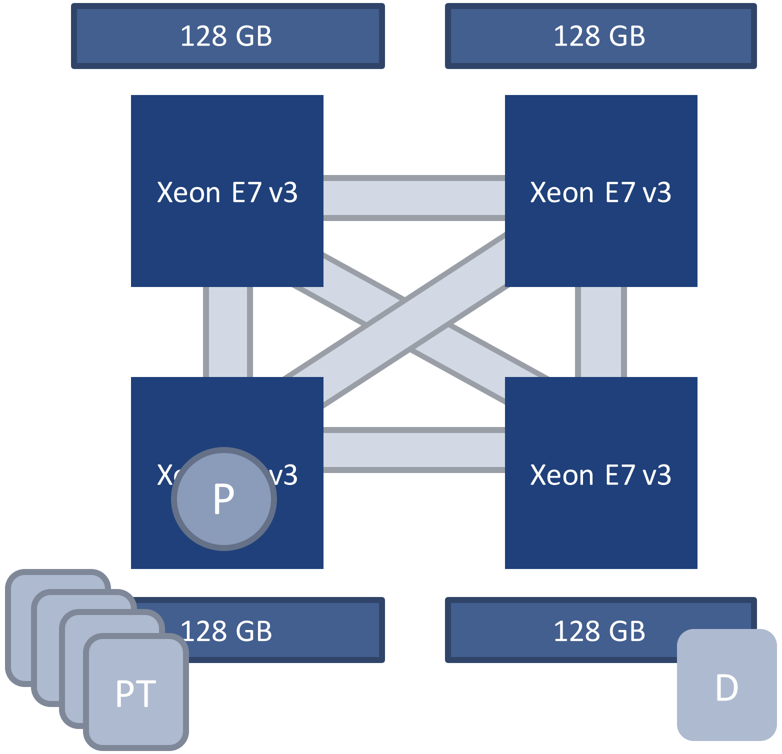
- HPCC RandomAccess, 1 Thread, 64G table size
- Perf to obtain the performance counters

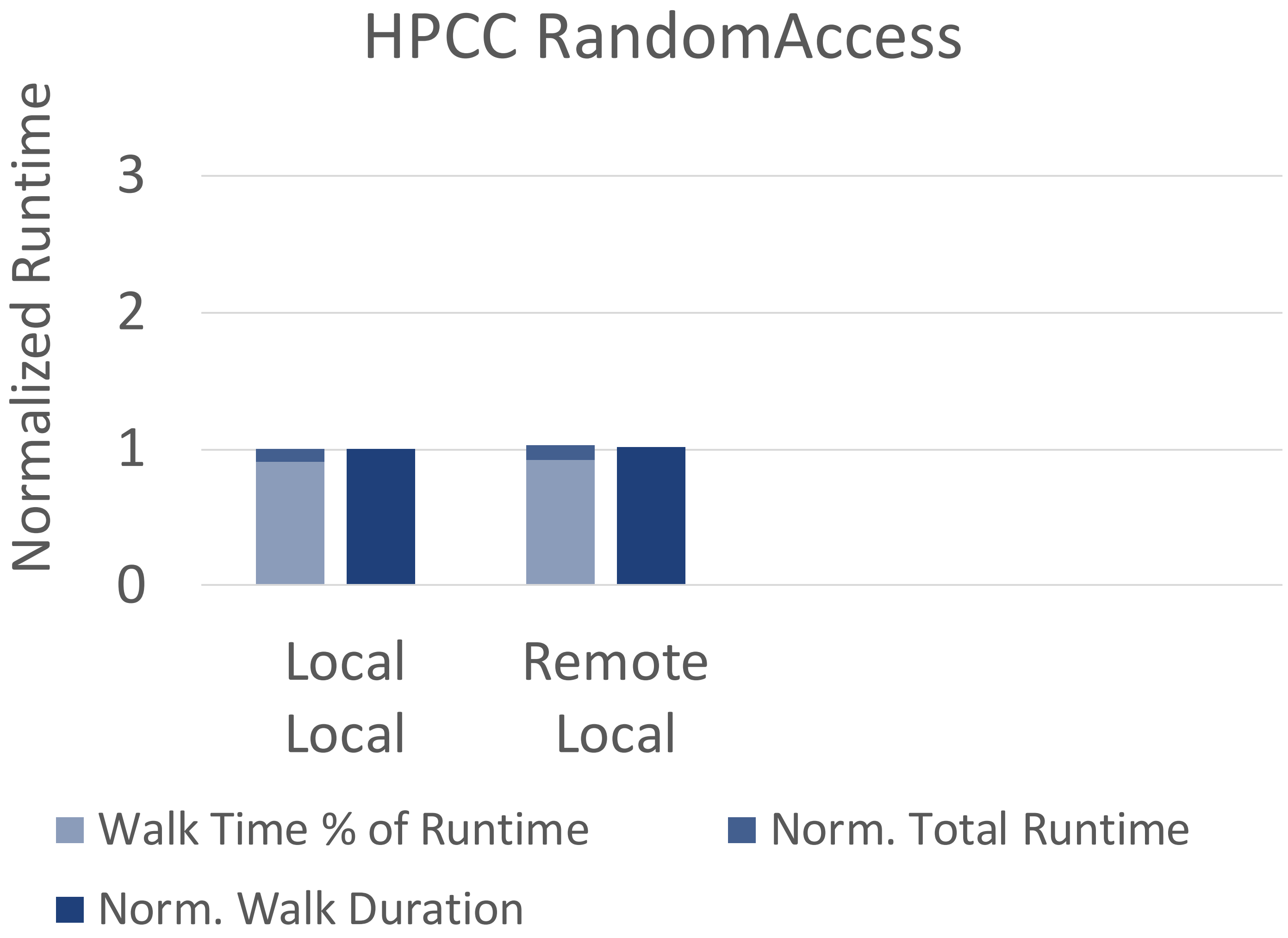
Effects of page-table Placements – Base Case: Data & Page Tables Local



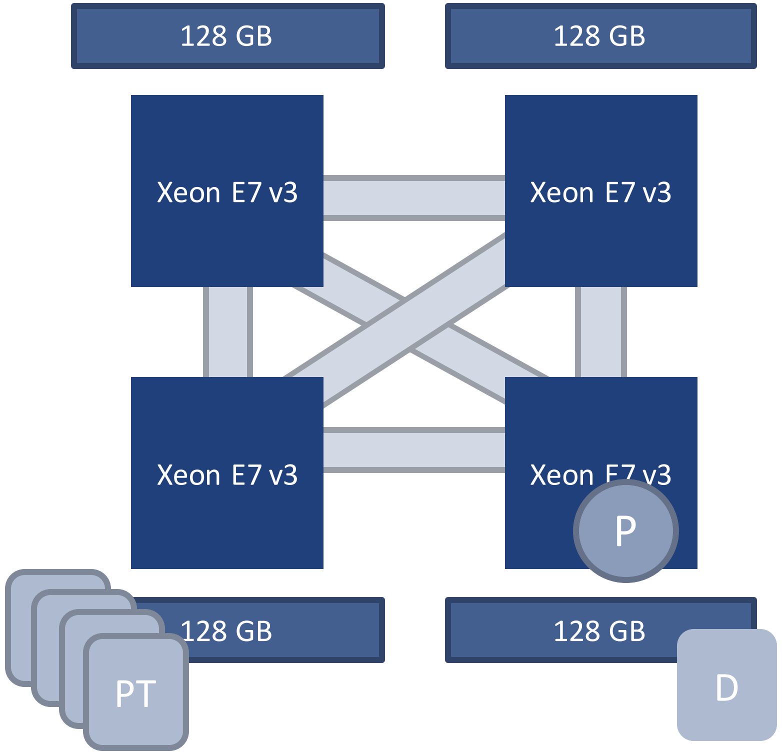


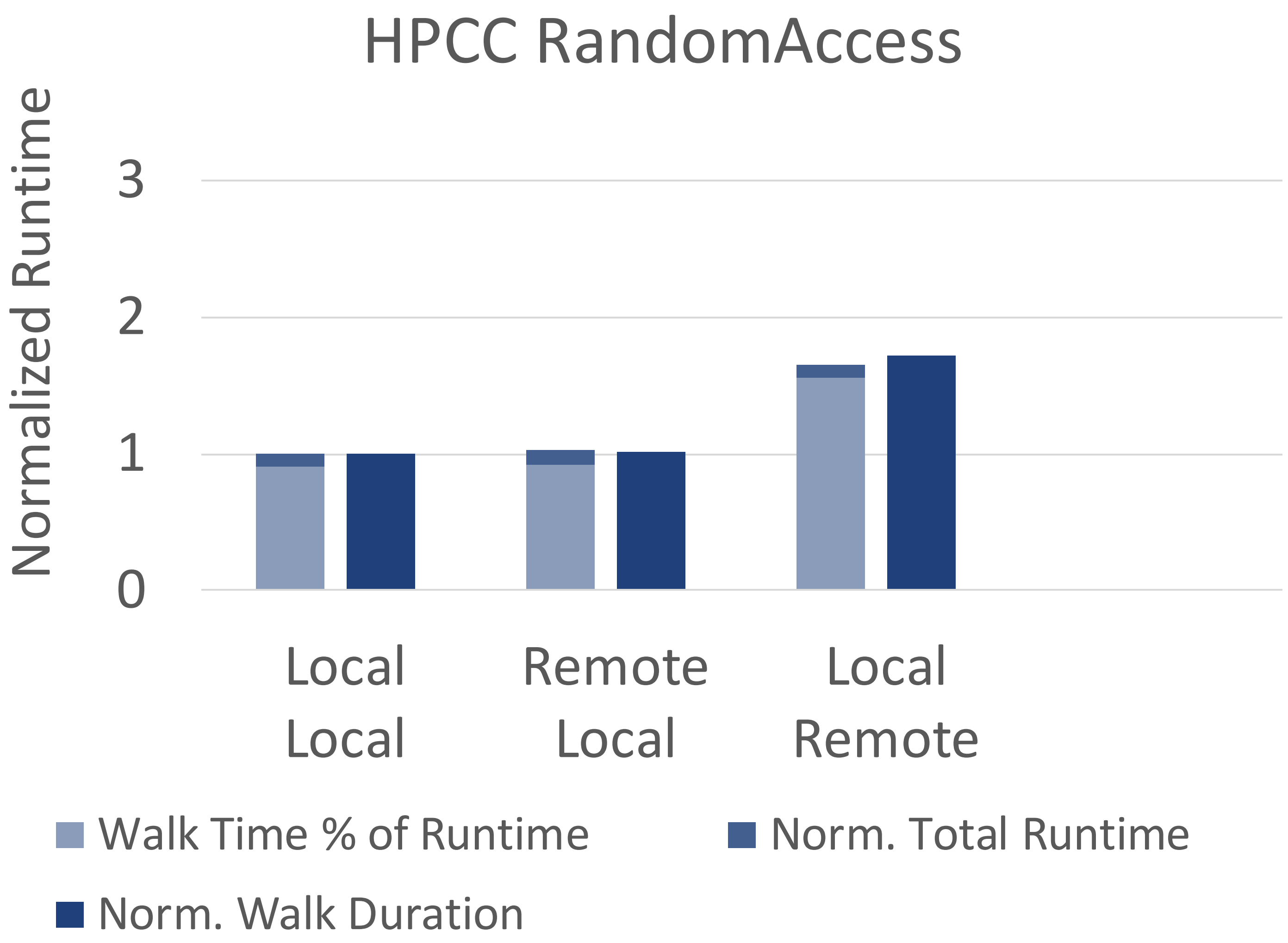
Effects of page-table Placements – Case 1: Data Remote / Page Tables Local



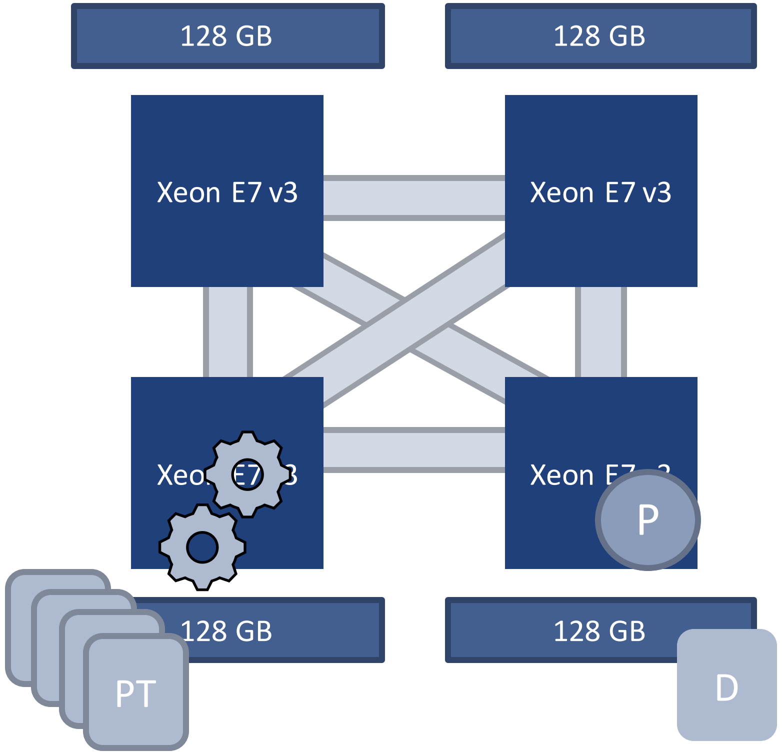


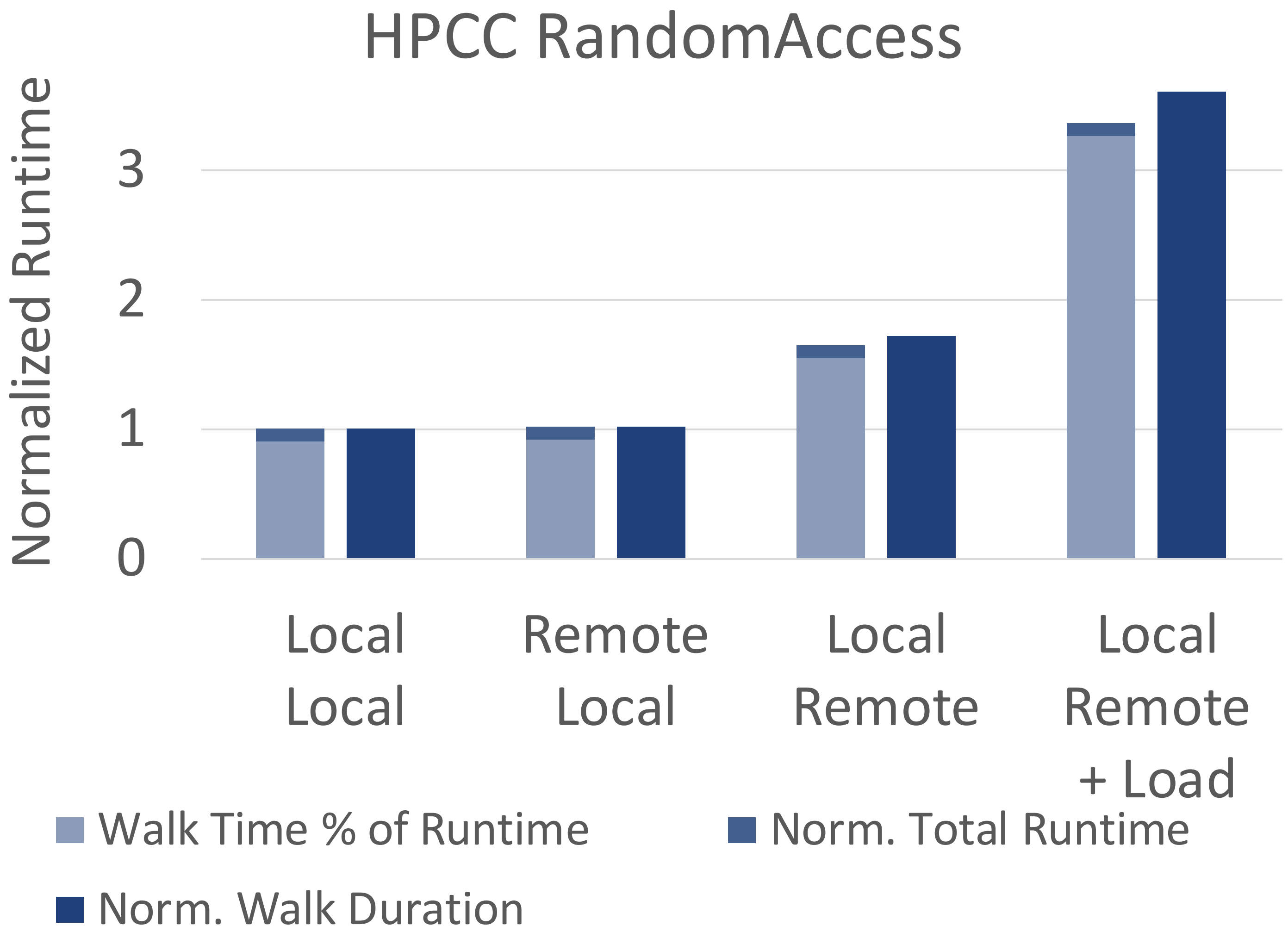
Effects of page-table Placements – Case 2: Data Local / Page Tables Remote





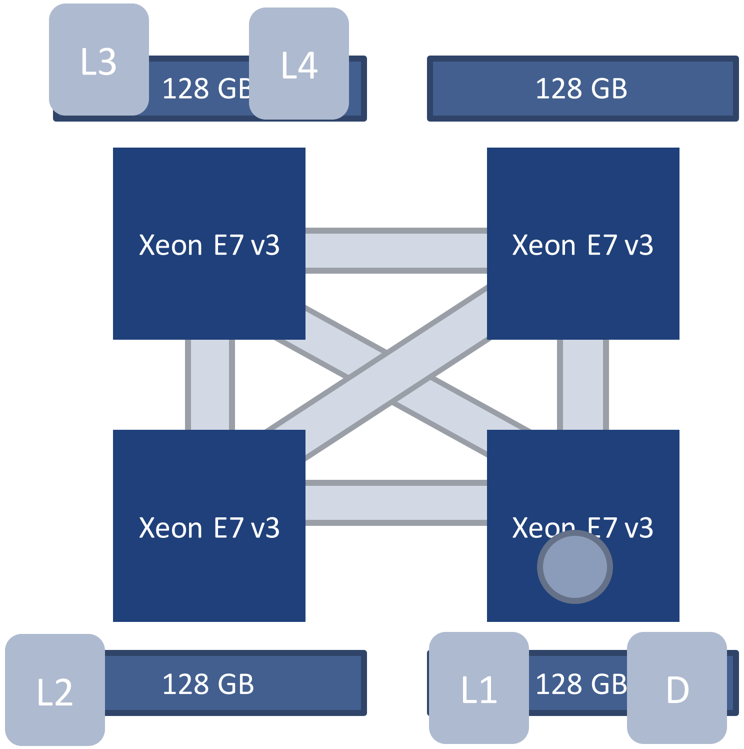
Effects of page-table Placements – Case 3: Loaded Page Table Node





How often happens the page table to be remote?

Page table allocation statistics - Multi-threaded Workloads



- How are the page tables allocated ?
- How do they change over time?
- Methodology:
  Let the workload run and dump the page table every 30s.
- Breakdown and diff between two dumps

```
$ ./pagerank hugegraph.bin –nthreads 112

PTablesLevel1   44k [11M  3M  3M  3M]   (+67,-0) | 24k [ 1M  8M  1M  1M]   (+66,-0) | 23k [ 1M  1M  8M  1M]   (+67,-0) | 24k [ 1M  1M  1M  8M]   (+64,-0) || 116k
PTablesLevel2    71 [24k  3k  3k  4k]    (+0,-0) |  90 [11k 12k 10k 10k]    (+0,-0) |  38 [ 3k  4k  4k  3k]    (+0,-0) |  37 [ 4k  3k  4k  5k]    (+0,-0) ||  236
PTablesLevel3     1 [ 24  84   9  35]    (+0,-0) |   0 [  0   0   0   0]    (+0,-0) |   3 [ 47   6  29   2]    (+0,-0) |   0 [  0   0   0   0]    (+0,-0) ||    4
PTablesLevel4     0 [  0   0   0   0]    (+0,-0) |   0 [  0   0   0   0]    (+0,-0) |   1 [  8   0   3   1]    (+0,-0) |   0 [  0   0   0   0]    (+0,-0) ||    1
Code2M            0 [ --  --  --  --]    (+0,-0) |   0 [ --  --  --  --]    (+0,-0) |   0 [ --  --  --  --]    (+0,-0) |   0 [ --  --  --  --]    (+0,-0) ||    0
Data2M            0 [ --  --  --  --]    (+0,-0) |   0 [ --  --  --  --]    (+0,-0) |   0 [ --  --  --  --]    (+0,-0) |   0 [ --  --  --  --]    (+0,-0) ||    0
Data4k           9M [ --  --  --  --] (+47k,-4M) |  9M [ --  --  --  --] (+48k,-4M) |  9M [ --  --  --  --] (+48k,-4M) | 10M [ --  --  --  --] (+48k,-4M) ||  38M
Code4k          338 [ --  --  --  --]    (+0,-0) | 439 [ --  --  --  --]    (+0,-0) | 176 [ --  --  --  --]    (+0,-0) |  80 [ --  --  --  --]    (+0,-0) ||   1k
NUMACode2M        0 [ --  --  --  --]    (+0,-0) |   0 [ --  --  --  --]    (+0,-0) |   0 [ --  --  --  --]    (+0,-0) |   0 [ --  --  --  --]    (+0,-0) ||    0
NUMAData2M        0 [ --  --  --  --]    (+0,-0) |   0 [ --  --  --  --]    (+0,-0) |   0 [ --  --  --  --]    (+0,-0) |   0 [ --  --  --  --]    (+0,-0) ||    0
NUMACode4k        0 [ --  --  --  --]    (+0,-0) |   0 [ --  --  --  --]    (+0,-0) |   0 [ --  --  --  --]    (+0,-0) |   0 [ --  --  --  --]    (+0,-0) ||    0
NUMAData4k       5M [ --  --  --  --]   (+4M,-1) |  5M [ --  --  --  --]   (+4M,-0) |  4M [ --  --  --  --]   (+4M,-0) |  5M [ --  --  --  --]   (+4M,-0) ||  20M
Total Migrationns 0   152
```

Local: 38%

Remote: 62%

Local: 20%

Remote: 80%

Local: 19%

Remote: 81%

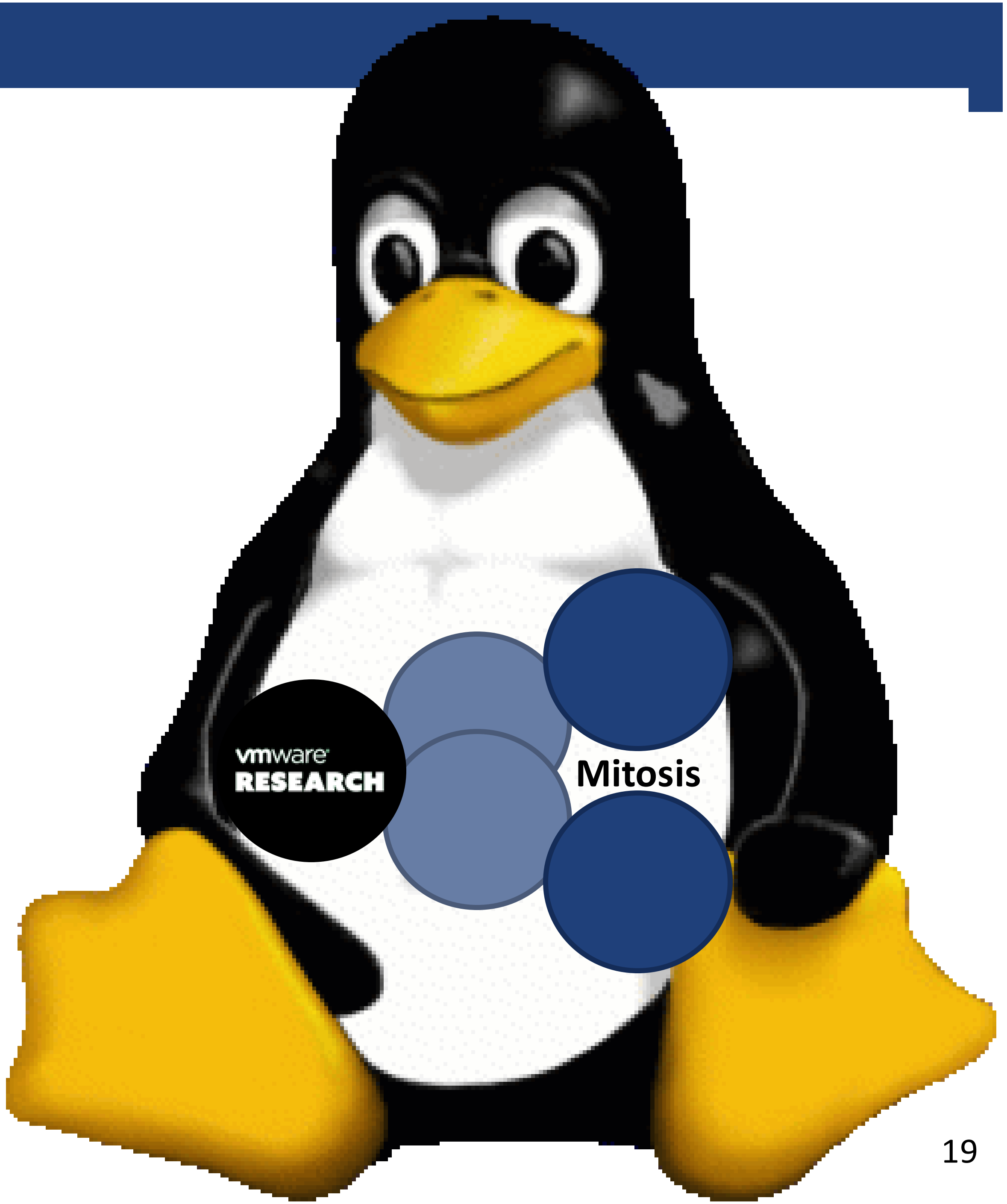
Local: 20%

Remote: 80%

# Remote > # Local

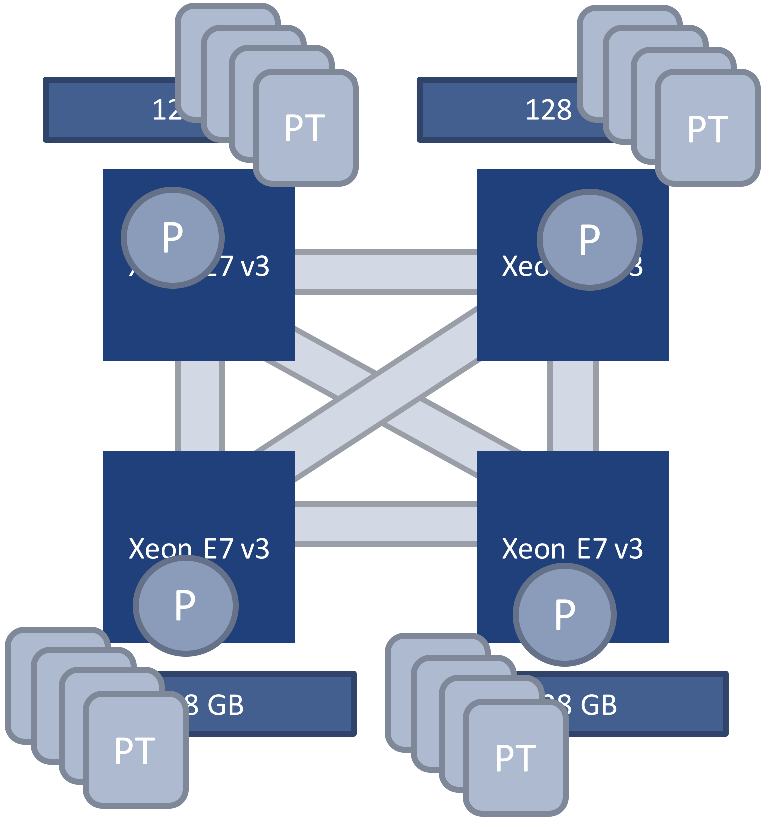
Page tables don’t move

> ¾ Remote!



Mitosis – Transparent self-replicating page tables on Linux / x86

Page Table Replication in a NUMA Machine



The key idea is keep page-tables local

Replication of page tables on each NUMA node

1. Native page tables for processes
2. Extended page tables for virtual machines

Keeping replicas consistent without sending the Kernel on an island to deal with the parliament there

Mitosis

Use the local replica on the current processor

Manage and find page table replicas efficiently

Keeping page table replicas consistent with each other

1) Programming the Translation Base Register (x86: cr3)

Reading

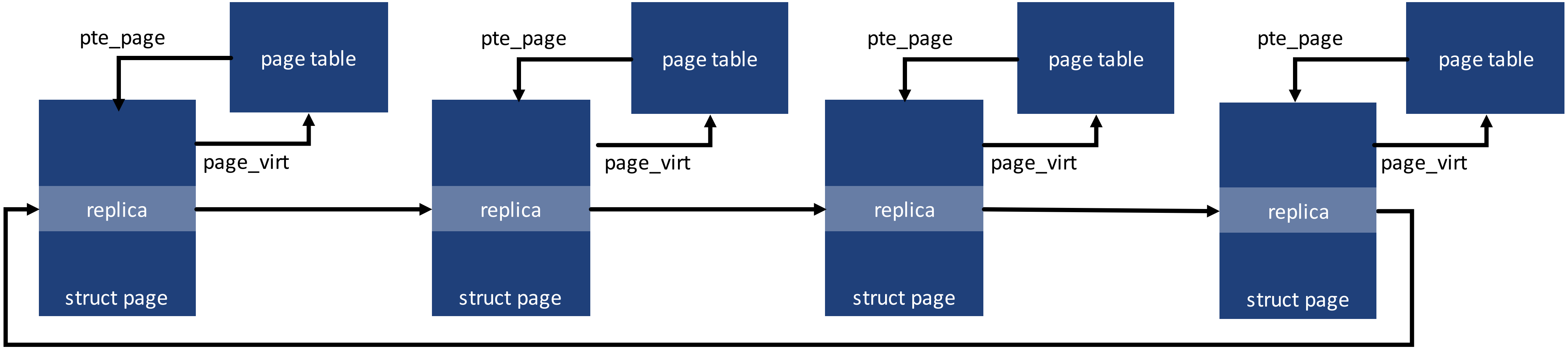
- Different CR3 values depending on the node you are running
- This may cause confusion in the kernel
- We look up the master replica and re-build the original CR3 value from it.

Writing

- Need to write the pointer to the **local page table** root
- Lookup the local page table and re-build the CR3 value from it.

2) Keeping Track of Replicas

- a page descriptor for each physical page
- Conversion functions: page table pointer ⇔ page descriptor pointer



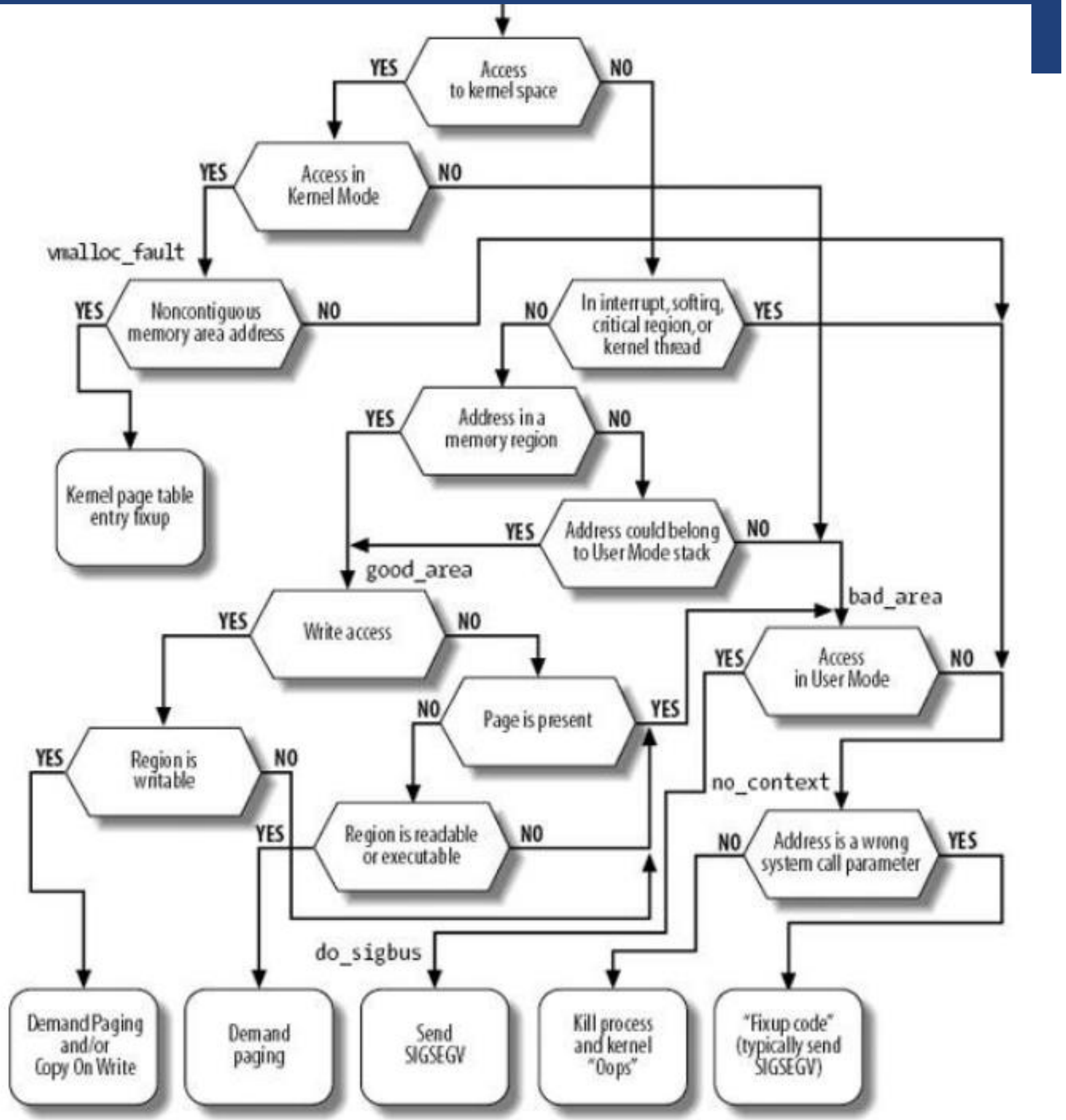
Add a pointer to the next replicas in the page descriptor

- Circular list of replicas

3) Keeping Page Tables Consistent --- Oh no….

- In a nutshell:
  1. find the containing vm_area_struct
  2. Check the permissions (read / write / exec)
  3. Walk page table and allocate missing tables
  4. Allocate a new frame & update the PTE
  5. Resume execution

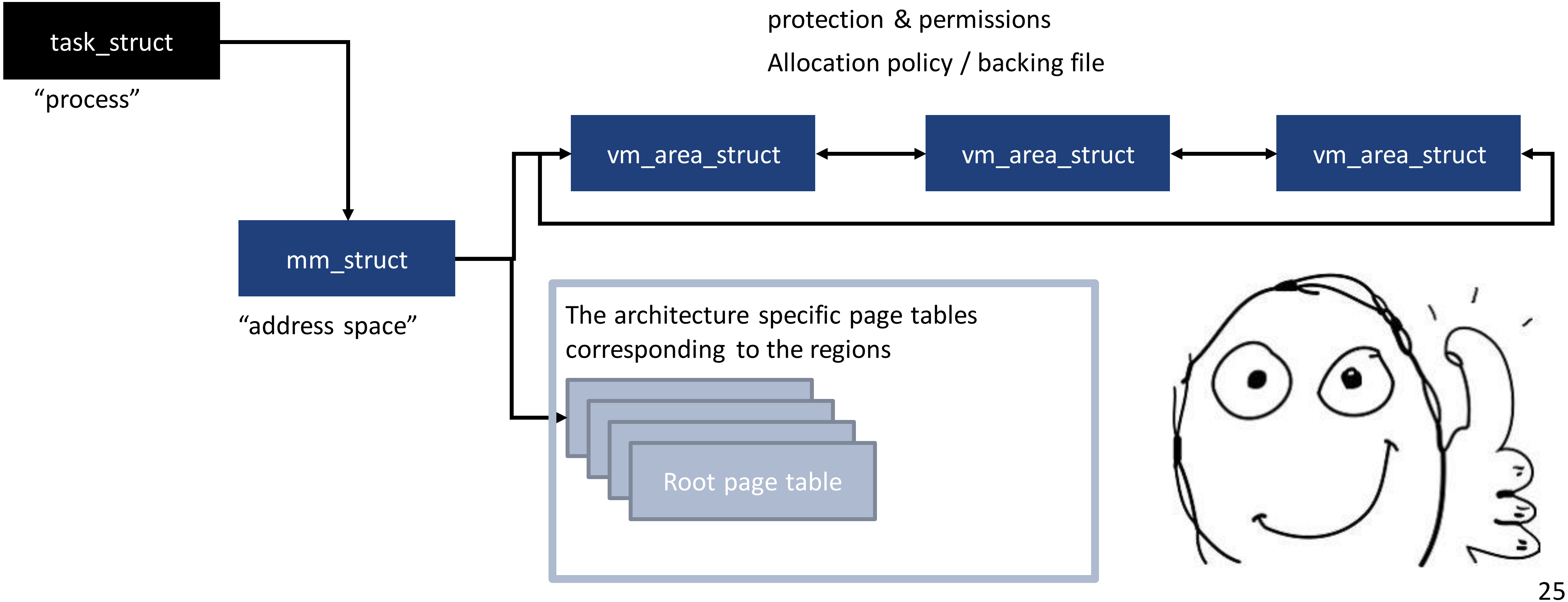
+ Deal with Copy-on-Write, NUMA balancing, huge pages, disk IO, NUMA policies, ….



Access to kernel space
YES
NO
Access in Kernel Mode
vmalloc_fault
Noncontiguous memory area address
In interrupt, softirq, critical region, or kernel thread
Address in a memory region
Kernel page table entry fixup
Address could belong to User Mode stack
good_area
bad_area
Write access
Page is present
Access in User Mode
Region is writable
Region is readable or executable
no_context
Address is a wrong system call parameter
do_sigbus
Demand Paging and/or Copy On Write
Demand paging
Send SIGSEGV
Kill process and kernel "Oops"
"Fixup code" (typically send SIGSEGV)

Understanding the Linux Kernel, Second Edition by Marco Cesati, Daniel P. Bovet

3) Linux Memory Management 10000ft View



Which regions have been allocated

protection & permissions

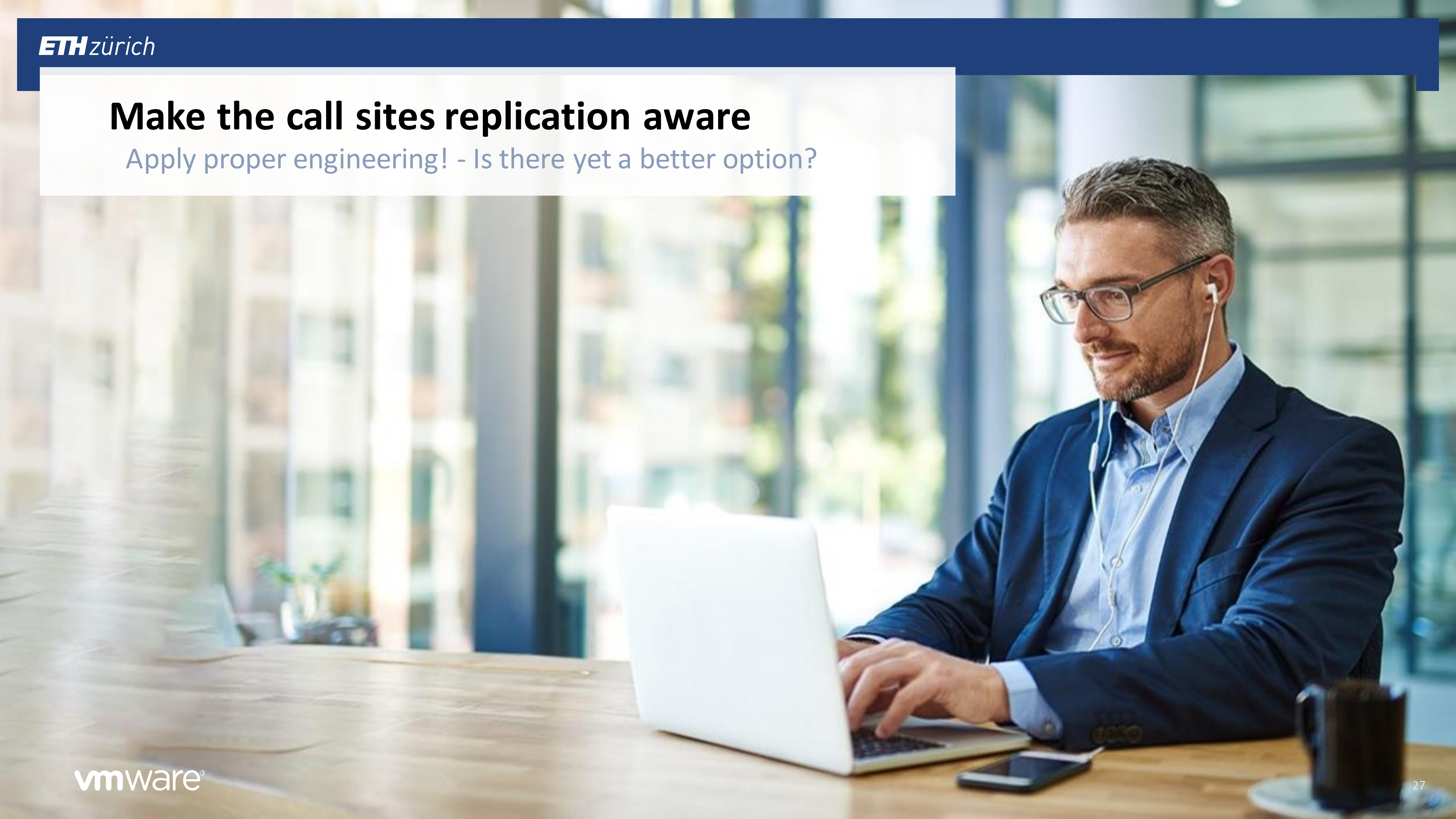
Allocation policy / backing file

The architecture specific page tables corresponding to the regions

Wait a second, isn't there a better option?

Make the call sites replication aware

Apply proper engineering! - Is there yet a better option?



PV-Ops: Para-Virtualization in the Linux Kernel

PV-Ops unified the kernel to run both, native and paravirtualized environments

A **table of function pointers** to native functions, or hypervisor calls for Xen, VMware VMI

• Allocation / deallocation of page tables of all levels	alloc_pte / release_pte
• Create / extract entries	make_pte / pte_val
• Set or clear entries in the page tables	set_pte / clear_pte
• Reads / writes to the CR3 register	read_cr3 / write_cr3

ONE DOES NOT SIMPLY
CHANGE LINUX' MEMORY SUBSYSTEM
imgflip.com

Mito

one!

PV-Ops: Modifications to the page table are handled, right ?

PV-Ops intercept

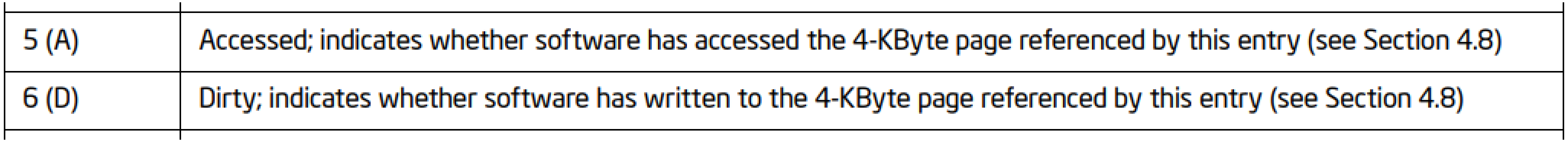
• Allocation / deallocation of page tables of all levels	alloc_pte / release_pte
• Create / extract entries	make_pte / pte_val
• Set or clear entries in the page tables	set_pte / clear_pte
• Reads / writes to the CR3 register	read_cr3 / write_cr3

PV-Ops don’t intercept

• Reads	*ptep / pte_write(*ptep)
• Writes in special occasions e.g. write protects	*orig_pte = *pte

And then things get dirty…

- Intel Architectures Software Developer’s Manual states



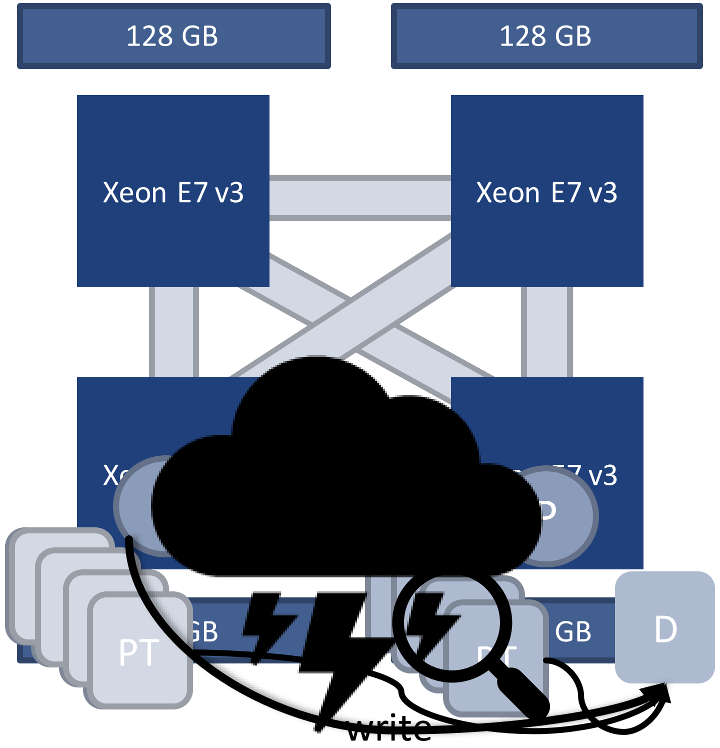
5 (A)	Accessed; indicates whether software has accessed the 4-KByte page referenced by this entry (see Section 4.8)
6 (D)	Dirty; indicates whether software has written to the 4-KByte page referenced by this entry (see Section 4.8)

- What’s the problem with this snippet

```
if (pte_young(*ptep) || pte_dirty(*ptep)) {
   // do something
}
```

- Used in file maps, write protection, NUMA balancing, swap entries, …

Access and dirty bits matter



1. Program allocates memory, Kernel faults in some anonymous RAM, updates all replicas

2. Program runs, writes to the allocated page

3. Kernel reads the entry for some policy mechanism

```
if (pte_young(*ptep) || pte_dirty(*ptep)) {
  // do something
}
```

4. Kernel doesn’t see the dirty / accessed bit, concludes wrong decision.

Reading Page Table Entries

Two possible cases

1. If the entry is a **leaf** then all replicas point to the **same page**.

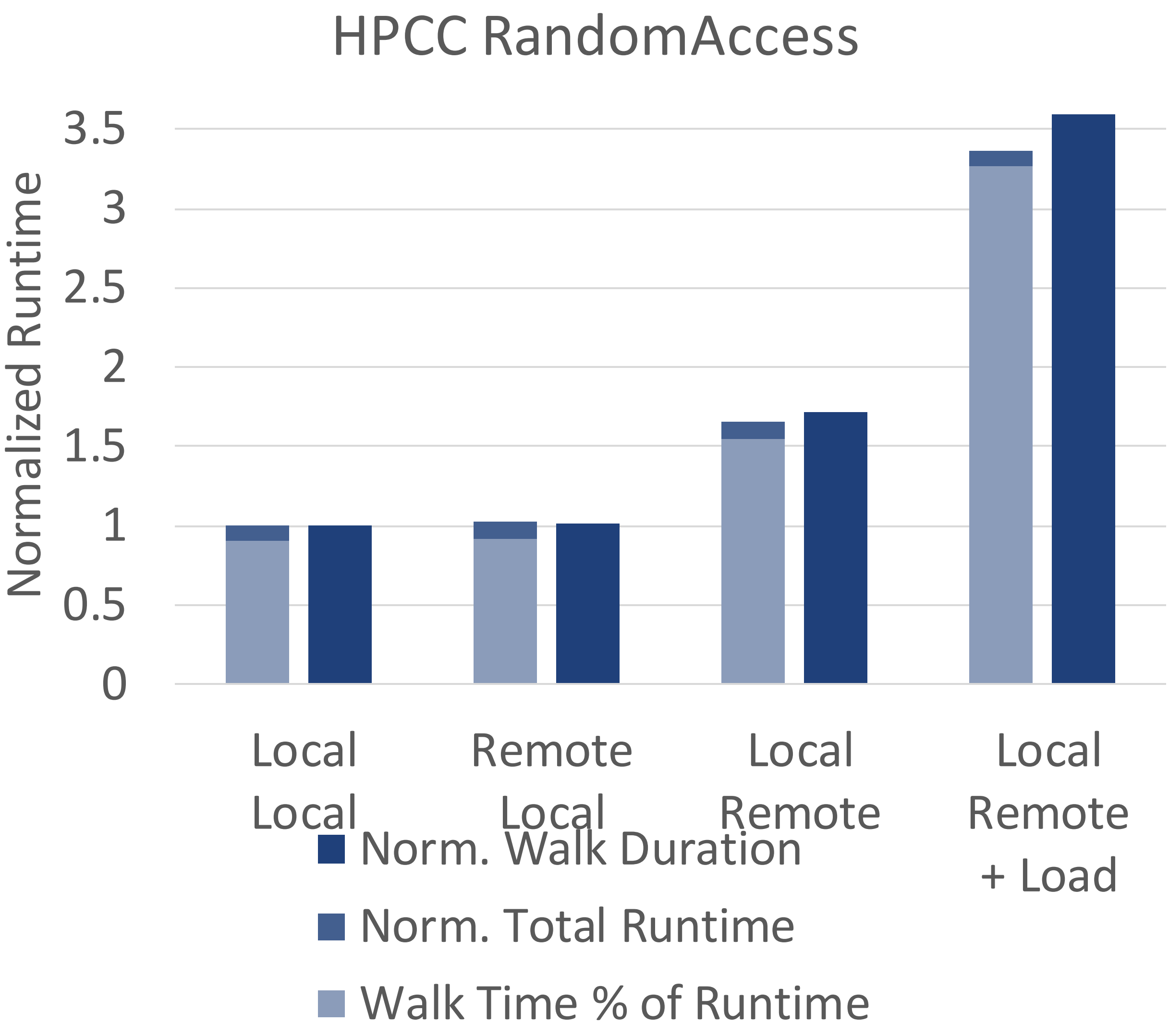
2. If the entry is **not a leaf**, then the entries point to **different page tables**!

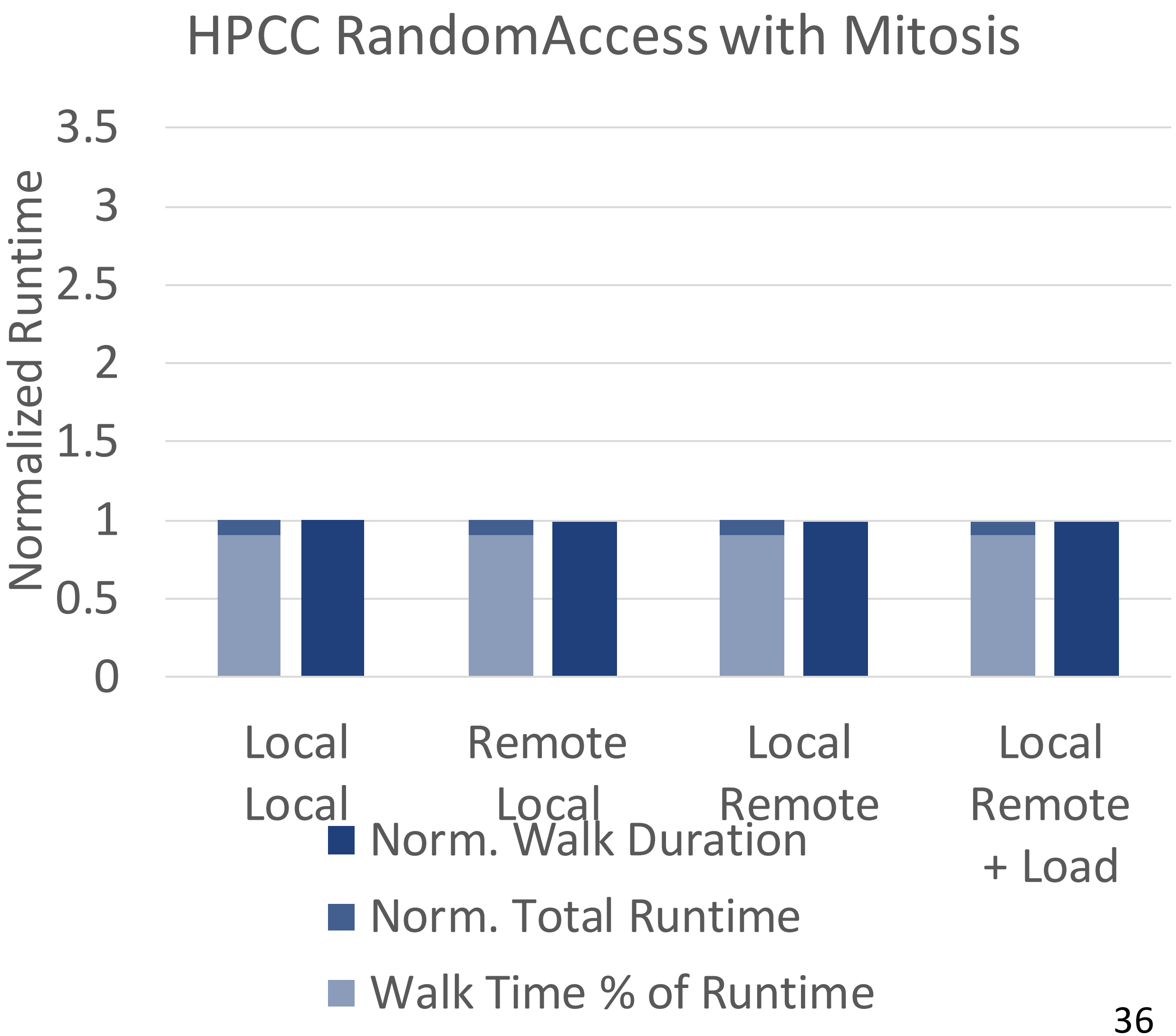
```
pte_t ptep_read (pte_t  *ptep)
{
    pte_t  pte = 0;
    FOREACH(pte_t *p : replicas(ptep) {
        pte |= *p;
    }
    return pte;
}
```

It’s actually what we had to do!

- Results

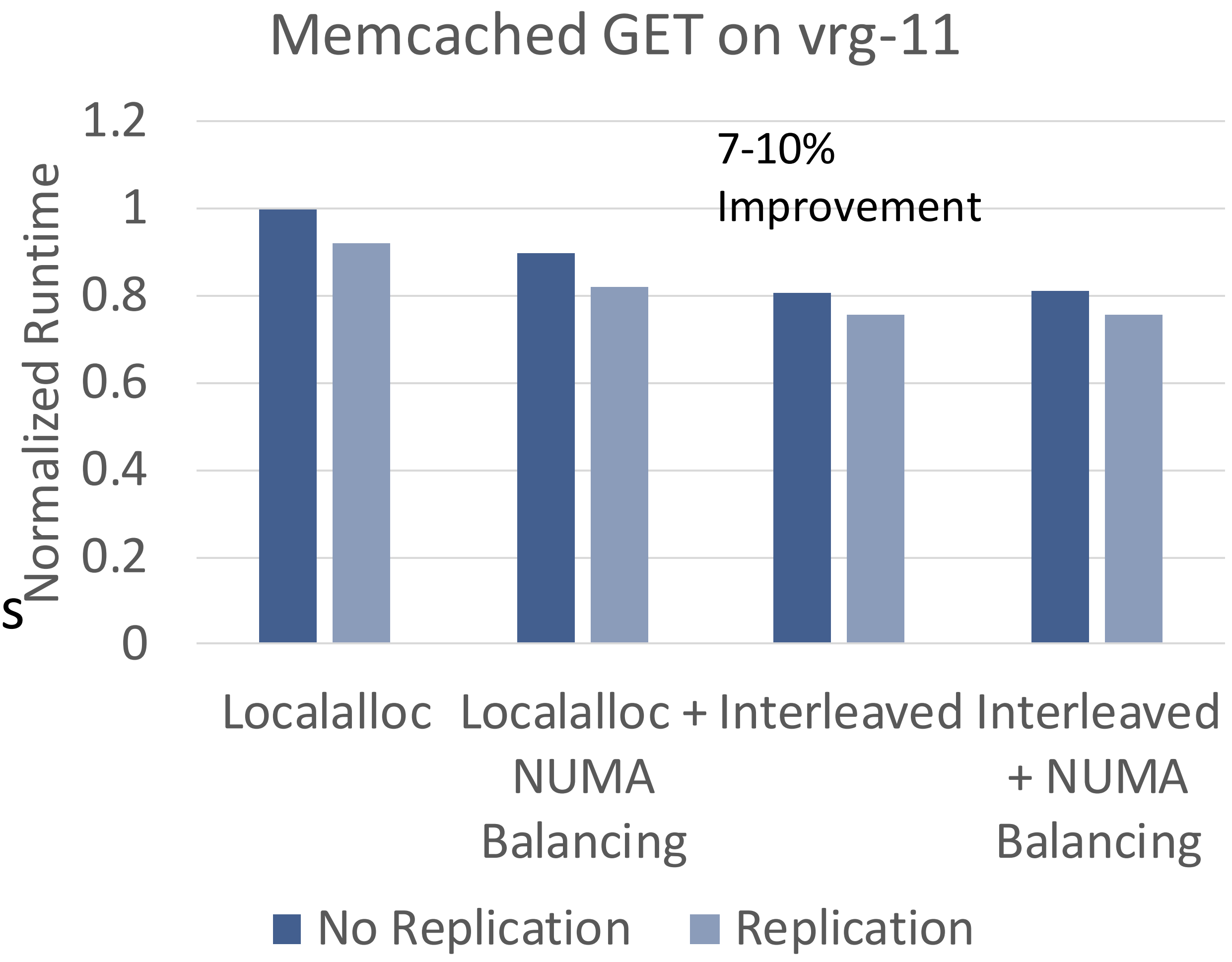
Results: Single Threaded Workloads





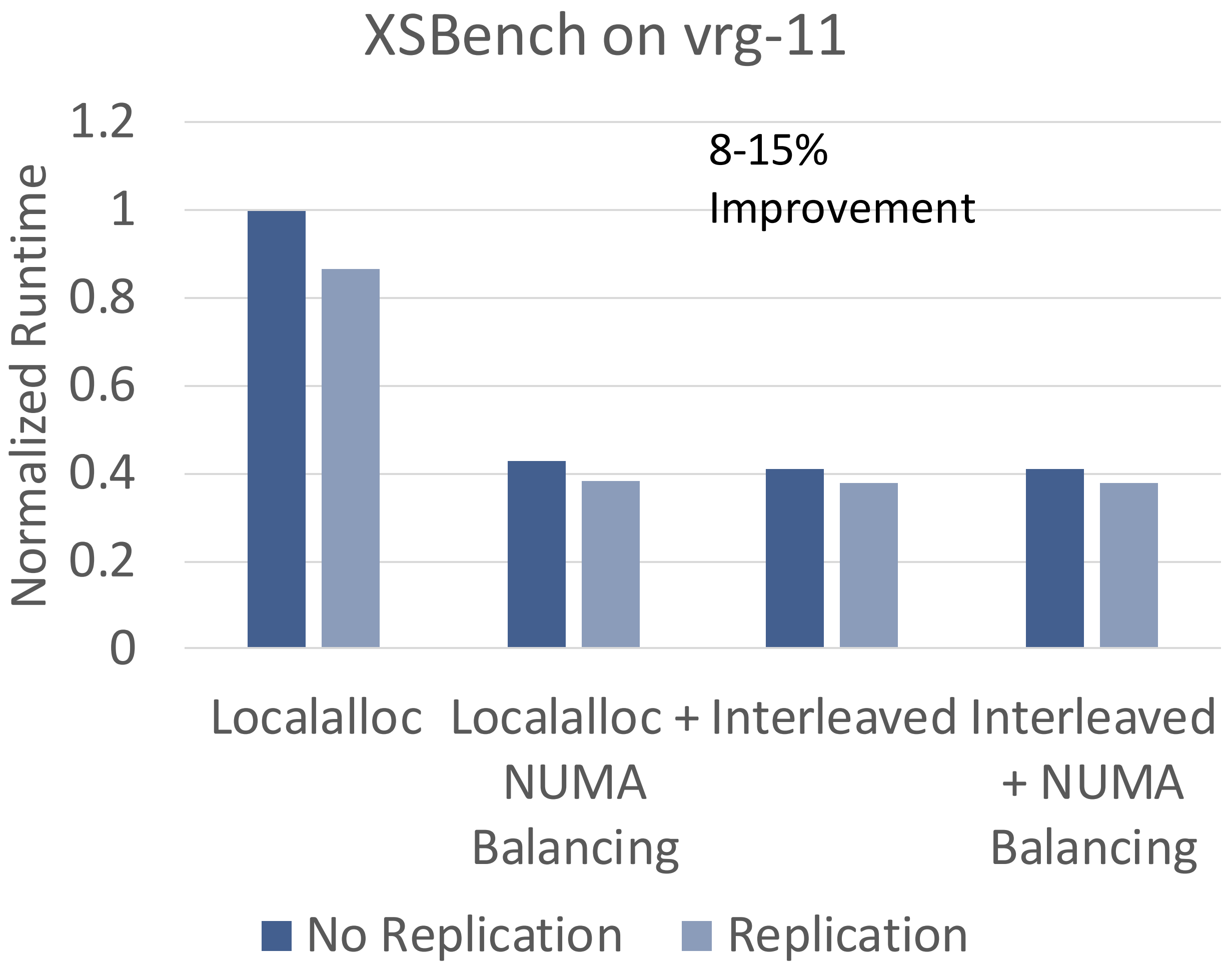
Results: Multi-threaded Workloads - Memcached

- Initialization (not profiled)
  - Pre-allocated SLABs
  - Population of the DB state.
- Benchmark
  - Parallel GET of randomly chosen keys
  - Accessed by 112 threads in parallel



- *“ The XSBench proxy app models the most computationally intensive part of a typical Monte Carlo transport algorithm”*

  Initialization (not profiled)
  - Allocation of the data structures
- Benchmark
  - Full XSBench Simulation



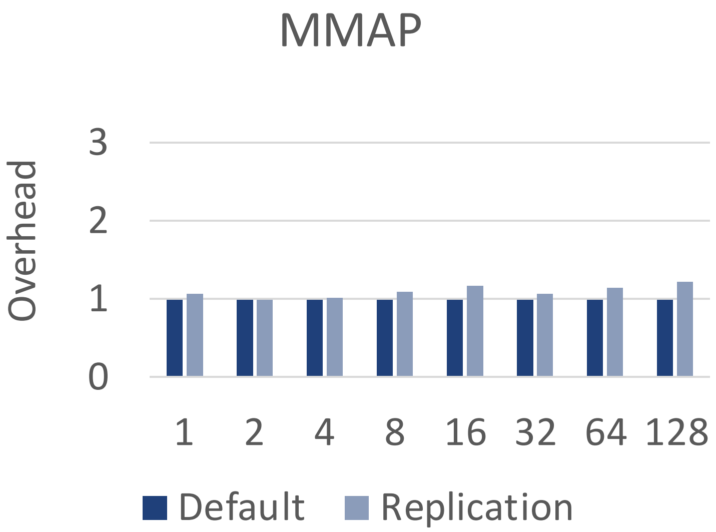
Overheads: Memory

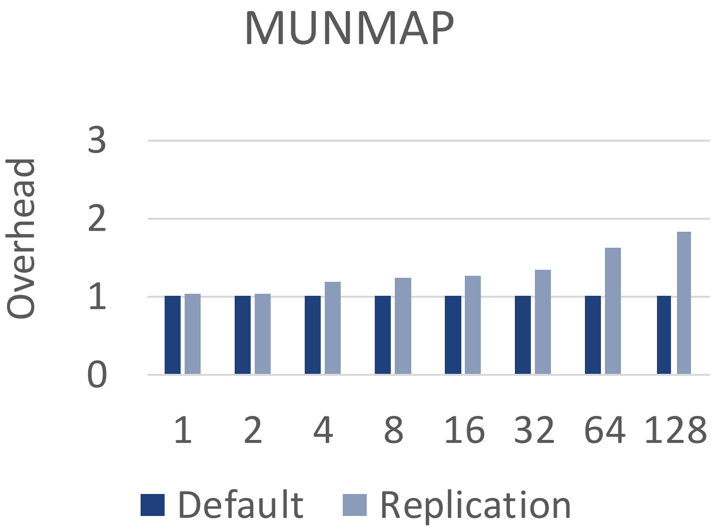
Memory overhead for page descriptor: it depends on kconfig. At most 0.4%

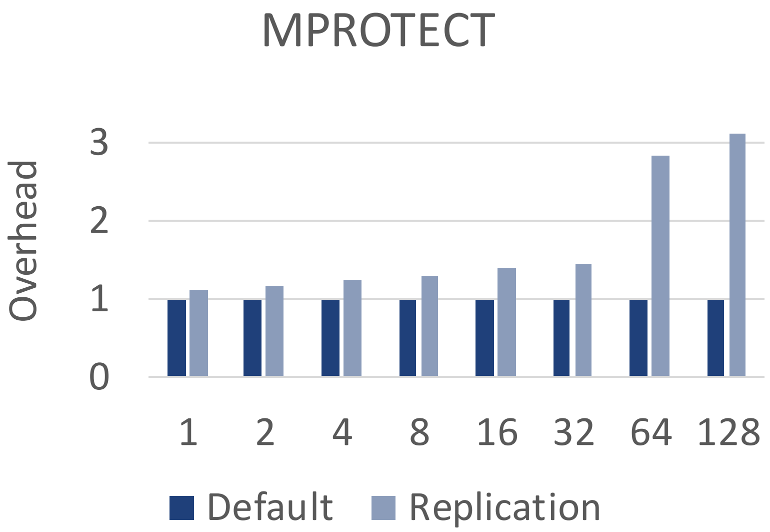
Memory Overhead of 400GB working set

1 Replica: (206k + 410 + 4 + 1) = 806MB (0.19%)
4 Replica: = 2418MB (+0.59%)

Overheads: Virtual Memory Operations







Future Work

Virtual machines use Extended / Nested Page Tables.

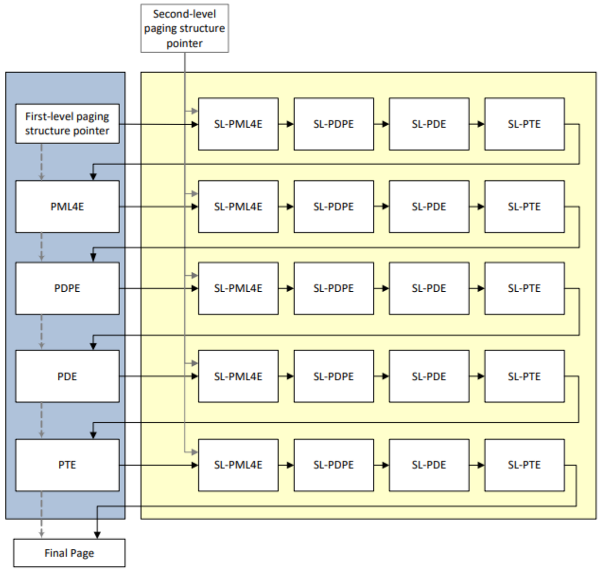
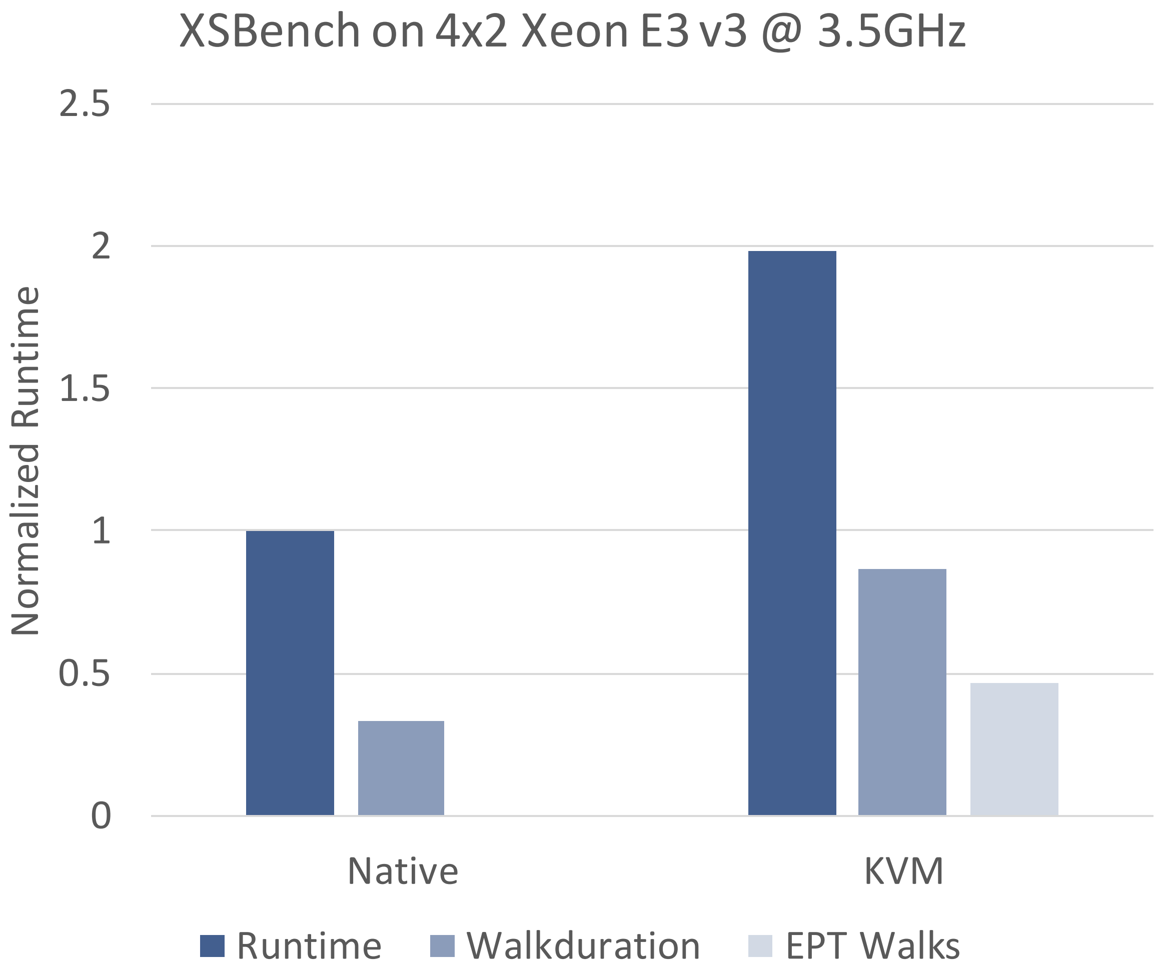
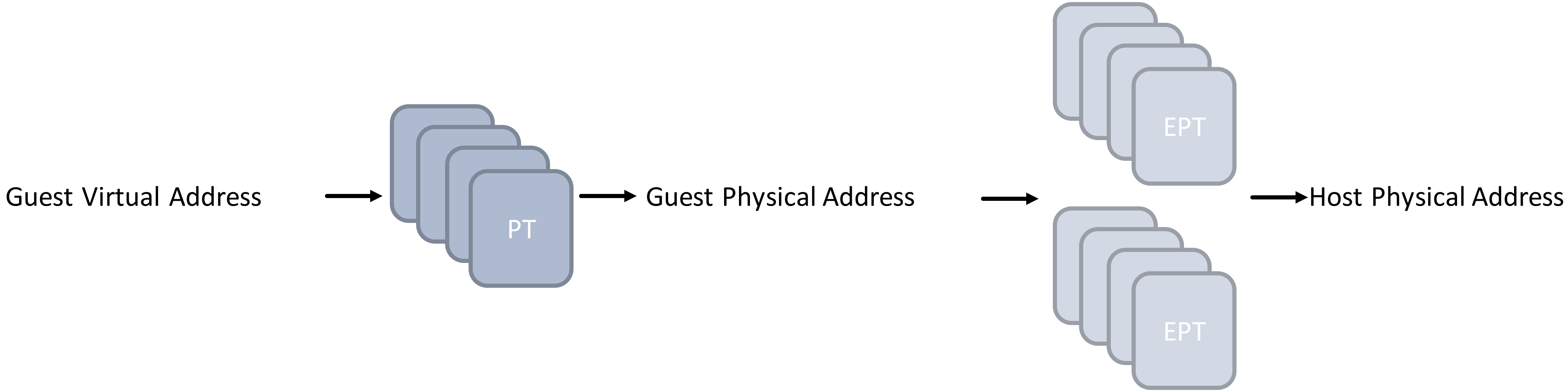


Figure 3-12. Nested Translation with 4-KByte pages

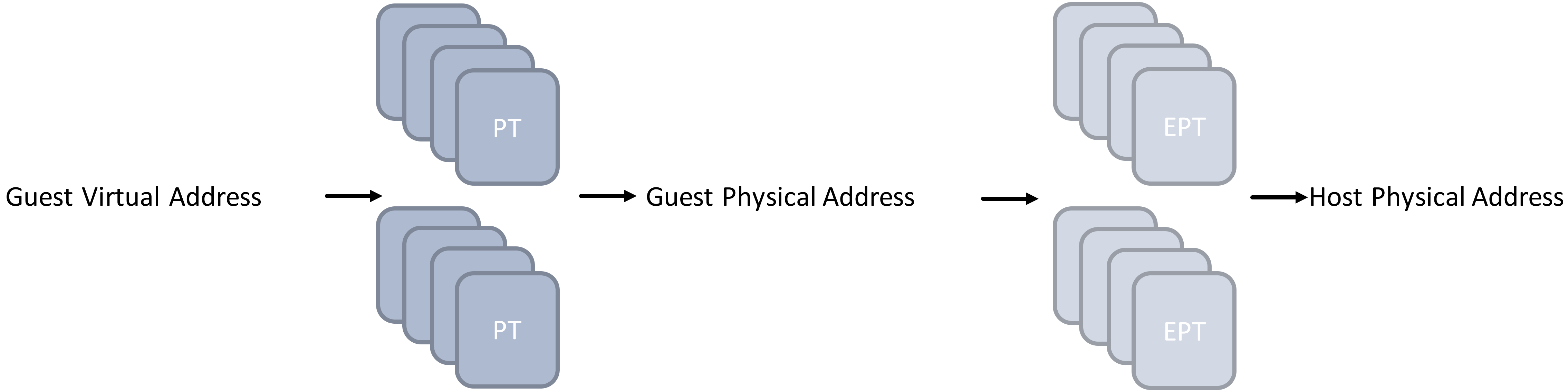
Up to 24 Memory accesses



Future Work: Hypervisor Implementation – EPT Only



Future Work: Coopereative Replication of EPT + Guest PT



Guest Virtual Address

PT

PT

Guest Physical Address

EPT

EPT

Host Physical Address

Conclusions

- Bad page table placement hurts the performance
- Mitosis avoids a 3.4x slowdown
- Speedup in several of workloads without modifications
- Promising applications in virtual machines