6. Memory Management

TLB, hit/miss, hit rate, temporal/spatial locality, TLB entry, ASID, multi-level page table, PTBR, PDBR, CR3, inverted page table, swap, page fault, minor/major page fault, optimal/fifo/random/LRU/clock page replacement, demand-paging vs pre-paging/pre-fetching, thrashing, working set, hugepages

To major issues with `Paging` are:

1. It is too slow, every memory access (also called a memory reference) leads to an extra memory access since the page table is stored in RAM; **Solved with TLB (translation lookaside buffer)**:
    - A part of the MMU
    - an *address-translation cache* that stores the most recent translations
    - Such that the MMU can look up the translation in the TLB, instead of the page table (faster)
2. The page table is too big (takes up too much space in RAM), let's solve this with one of
    - Multi-level page table (most used)
    - Inverted page table

6.1.1 TLB (Translation Lookaside Buffer)

Example:

```
int i, sum = 0;
  for (i = 0; i < 10; i++) {
  sum += a[i];
}
```

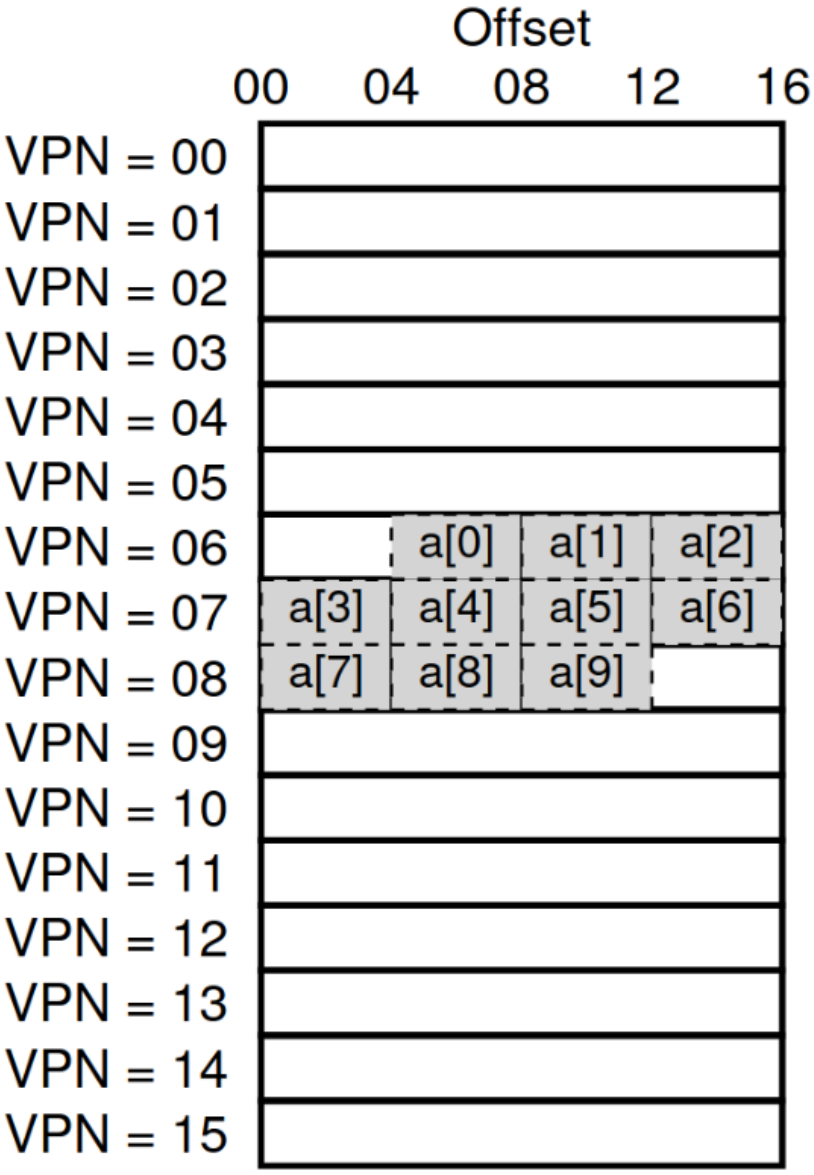


Figure 19.2: **Example: An Array In A Tiny Address Space**

Elements that reside besides each other will get faster access times, because of the TLB-lookup.

Thus, our **TLB hit-rate** is 70% (7 out of 10 hits).

To take advantage of hardware caches, we need to exploit *locality* (in instructions and references).

- **Temporal locality**: If you access a memory location, you are likely to access it again soon.
- **Spatial locality**: If you access a memory location, you are likely to access nearby memory locations soon.

TLBs rely on both temporal and spatial locality to achieve high hit rates.

Who needs to handle TLB misses?

In old days, the hardware would handle it:

1. It needs to know *where* the page table (via **page table register**) is stored in memory, and their *exact format*.
2. "Walk" the page table to find the correct entry and extract, and update TLB.

Modern CPUs use a **Hardware-managed TLB**:

1. The hardware raises an exception when a TLB miss occurs.
2. -> interrupt handler in the OS kernel is invoked to handle the miss.
3. the code will look up the correct entry in the page table and update the TLB.
4. and return

Advantages of this approach:

- The OS can decide how to handle TLB misses -- **flexibility**.
- The hardware doesn't do much -- **simplicity**.

6.1.2 TLB contents

Typical entry:

VPN | PPN | Other bits

- Other bits:
  - Valid bit: Is this entry valid traslation?
  - Protection bits: Can we read/write/execute this page?
  - etc.

When problem arises with TLB: **HOW TO MANAGE TLB CONTENTS ON A CONTEXT SWITCH?**

- Simply solution is to *flush* the TLB on a context switch (setting all valid bits to 0 thus clearing out the TLB).

But flushing could introduce a performance penalty, so we have a better solution:

ASID (Address Space Identifier)

VPN	PFN	valid	prot	ASID
10	100	1	rwx	1
—	—	0	—	—
10	170	1	rwx	2
—	—	0	—	—

6.2 Smaller Page Tables

Trying to tackle huge memory usage.

Bigger pages lead to internal fragmentation, *smaller pages* lead to more page table entries.

6.2.1 Multi-level Page Table

Idea:

1. Chop up the page table into page-sized units.
2. Then, if an entire page (of page-table entries PTEs) is invalid, don't allocate that page of the page table at all
3. To track whether a page is valid or not, use a new structure called **Page Directory**.
4. The page directory thus either can be used to tell you where a page of the page table is, or that the entire page of the page table contains n valid pages

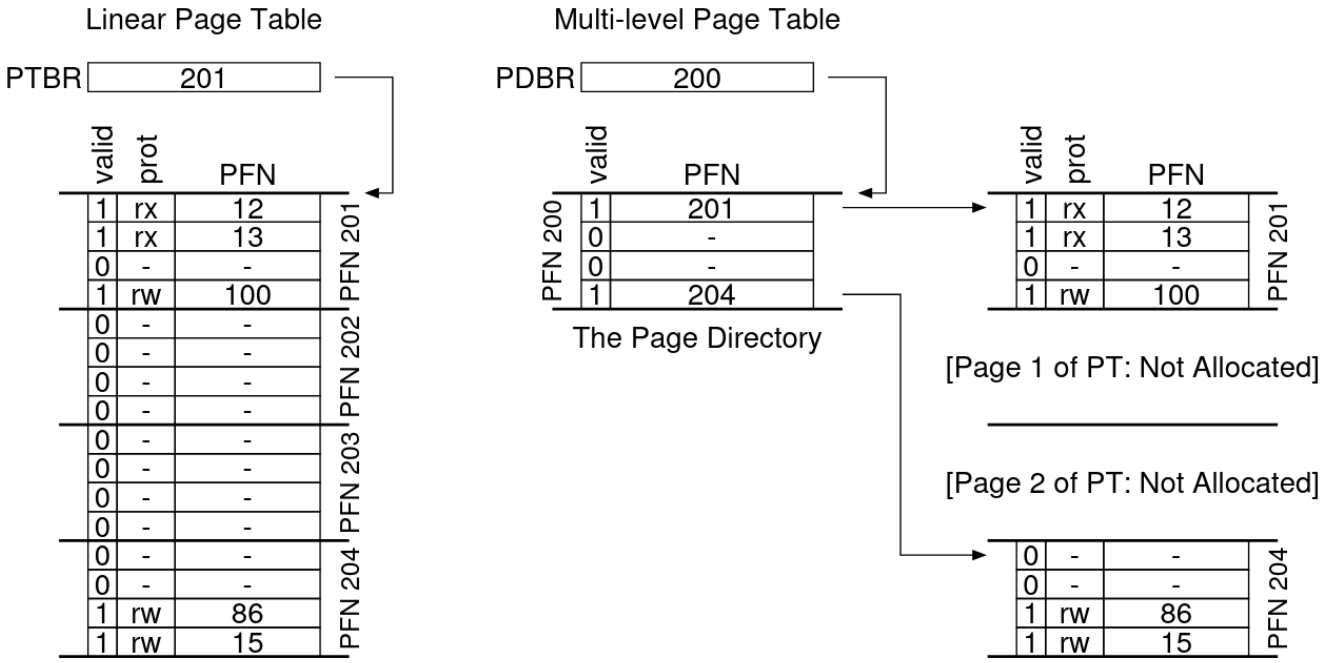


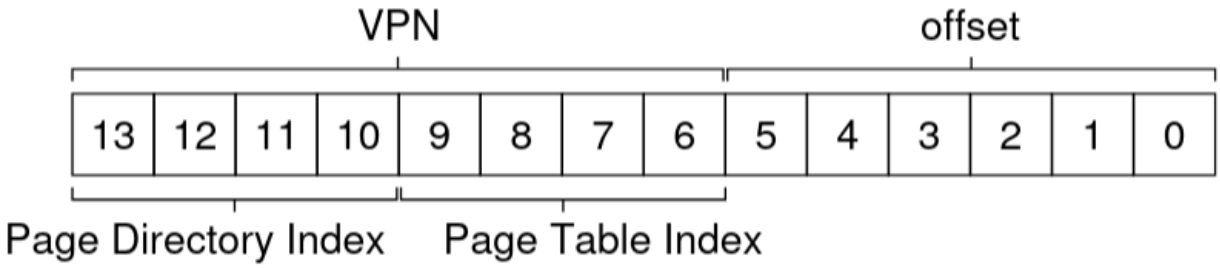
Figure 20.3: **Linear (Left) And Multi-Level (Right) Page Tables**

The page directory consist:

- a number of **page directory entries** (PDEs)
    - A PDE has a **valid bit** and a **Page frame number** (PFN) similar to a PTE.
    - however, the meaning of *valid bit* is different here; it tells you whether the PDE points to a (at least) page of the page table or not. Multi-level pages have some advantages:
- **Compact**: only allocate page table entries when needed
- Each page fits nicely

Disadvantages:

- On a TLB miss two memory accesses are needed to find the PTE
- complexity




```
1  VPN = (VirtualAddress & VPN_MASK) >> SHIFT
2  (Success, TlbEntry) = TLB_Lookup(VPN)
3  if (Success == True)    // TLB Hit
4    if (CanAccess(TlbEntry.ProtectBits) == True)
5      Offset   = VirtualAddress & OFFSET_MASK
6      PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
7      Register = AccessMemory(PhysAddr)
8    else
9      RaiseException(PROTECTION_FAULT)
10 else                    // TLB Miss
11   // first, get page directory entry
12   PDIndex = (VPN & PD_MASK) >> PD_SHIFT
13   PDEAddr = PDBR + (PDIndex * sizeof(PDE))
14   PDE     = AccessMemory(PDEAddr)
15   if (PDE.Valid == False)
16     RaiseException(SEGMENTATION_FAULT)
17   else
18     // PDE is valid: now fetch PTE from page table
19     PTIndex = (VPN & PT_MASK) >> PT_SHIFT
20     PTEAddr = (PDE.PFN<<SHIFT) + (PTIndex*sizeof(PTE))
21     PTE     = AccessMemory(PTEAddr)
22     if (PTE.Valid == False)
23       RaiseException(SEGMENTATION_FAULT)
24     else if (CanAccess(PTE.ProtectBits) == False)
25       RaiseException(PROTECTION_FAULT)
26     else
27       TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits)
28       RetryInstruction()
```

Figure 20.6: **Multi-level Page Table Control Flow**

See theory about two levels.

6.2.2 Inverted Page Table

> Here, instead of having many page tables (one per process of the system), we keep a single page table that has an entry for each physical page of the system

Cannot lookup, have to search the table for the entry..

6.3 Swap Space

> When we run out of physical memory, we need to swap some pages out to disk.

First we need to **reserve** some space on the disk for this purpose, this is called the **swap space**. Thus the OS will need to remember the **disk address** of a given page.

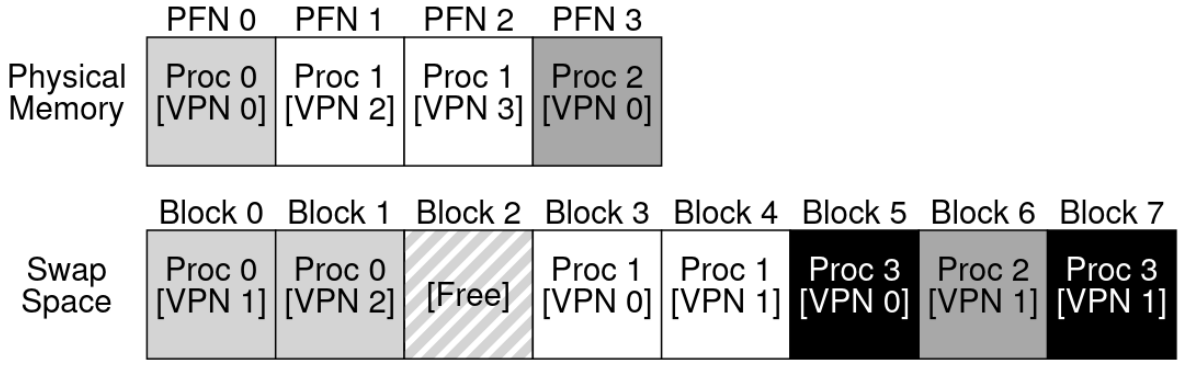


Figure 21.1: **Physical Memory and Swap Space**

Page fault: when a process tries to access a page that is not in physical memory.

In that case, the OS will invoke **page fault handler**.

So how doe the OS know where to find the page on disk?:

1. It looks in the PTE for the disk address and issues I/O to disk
2. When the I/O is complete, the OS updates the PTE and the TLB (while the process is in *blocked* state)

```
VPN = (VirtualAddress & VPN_MASK) >> SHIFT
(Success, TlbEntry) = TLB_Lookup(VPN)
if (Success == True)   // TLB Hit
    if (CanAccess(TlbEntry.ProtectBits) == True)
        Offset   = VirtualAddress & OFFSET_MASK
        PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
        Register = AccessMemory(PhysAddr)
    else
        RaiseException(PROTECTION_FAULT)
else                    // TLB Miss
    PTEAddr = PTBR + (VPN * sizeof(PTE))
    PTE = AccessMemory(PTEAddr)
    if (PTE.Valid == False)
        RaiseException(SEGMENTATION_FAULT)
    else
        if (CanAccess(PTE.ProtectBits) == False)
            RaiseException(PROTECTION_FAULT)
        else if (PTE.Present == True)
            // assuming hardware-managed TLB
            TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits)
            RetryInstruction()
        else if (PTE.Present == False)
            RaiseException(PAGE_FAULT)
```

Figure 21.2: **Page-Fault Control Flow Algorithm (Hardware)**

The process of picking a page to kick out, or replace is known as **page replacement policy**.

```
PFN = FindFreePhysicalPage()
if (PFN == -1)              // no free page found
    PFN = EvictPage()       // replacement algorithm
DiskRead(PTE.DiskAddr, PFN) // sleep (wait for I/O)
PTE.present = True          // update page table:
PTE.PFN     = PFN           // (present/translation)
RetryInstruction()          // retry instruction
```

Figure 21.3: **Page-Fault Control Flow Algorithm (Software)**

The OS ensure that there are pages higher than **Low watermark** but lower than **High watermark**. Swap daemon is responsible for this.

6.4 Page Replacement Policies

cache misses that is, when we need to fetch a page from disk.

Optimal policy: Replace the page that will not be used for the longest time in the future.

Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss		0
1	Miss		0, 1
2	Miss		0, 1, 2
0	Hit		0, 1, 2
1	Hit		0, 1, 2
3	Miss	2	0, 1, 3
0	Hit		0, 1, 3
3	Hit		0, 1, 3
1	Hit		0, 1, 3
2	Miss	3	0, 1, 2
1	Hit		0, 1, 2

Figure 22.1: **Tracing The Optimal Policy**

Optimal is impossible to implement in practice, but it is useful for comparison.

6.4.1 FIFO (First-In-First-Out)

Access	Hit/Miss?	Evict	Resulting Cache State	
0	Miss		First-in→	0
1	Miss		First-in→	0, 1
2	Miss		First-in→	0, 1, 2
0	Hit		First-in→	0, 1, 2
1	Hit		First-in→	0, 1, 2
3	Miss	0	First-in→	1, 2, 3
0	Miss	1	First-in→	2, 3, 0
3	Hit		First-in→	2, 3, 0
1	Miss	2	First-in→	3, 0, 1
2	Miss	3	First-in→	0, 1, 2
1	Hit		First-in→	0, 1, 2

Figure 22.2: **Tracing The FIFO Policy**

FIFO is simple to implement but not very good in practice.

6.4.2 Random

Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss		0
1	Miss		0, 1
2	Miss		0, 1, 2
0	Hit		0, 1, 2
1	Hit		0, 1, 2
3	Miss	0	1, 2, 3
0	Miss	1	2, 3, 0
3	Hit		2, 3, 0
1	Miss	3	2, 0, 1
2	Hit		2, 0, 1
1	Hit		2, 0, 1

Figure 22.3: **Tracing The Random Policy**

Random is simple to implement but not very good in practice.

6.4.3 LRU (Least Recently Used)

Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss		LRU→ 0
1	Miss		LRU→ 0, 1
2	Miss		LRU→ 0, 1, 2
0	Hit		LRU→ 1, 2, 0
1	Hit		LRU→ 2, 0, 1
3	Miss	2	LRU→ 0, 1, 3
0	Hit		LRU→ 1, 3, 0
3	Hit		LRU→ 1, 0, 3
1	Hit		LRU→ 0, 3, 1
2	Miss	0	LRU→ 3, 1, 2
1	Hit		LRU→ 3, 2, 1

Figure 22.5: **Tracing The LRU Policy**

LRU is hard to implement in practice but is very good.

NOTE: That is generates a new cache state each access

6.4.4 Workload

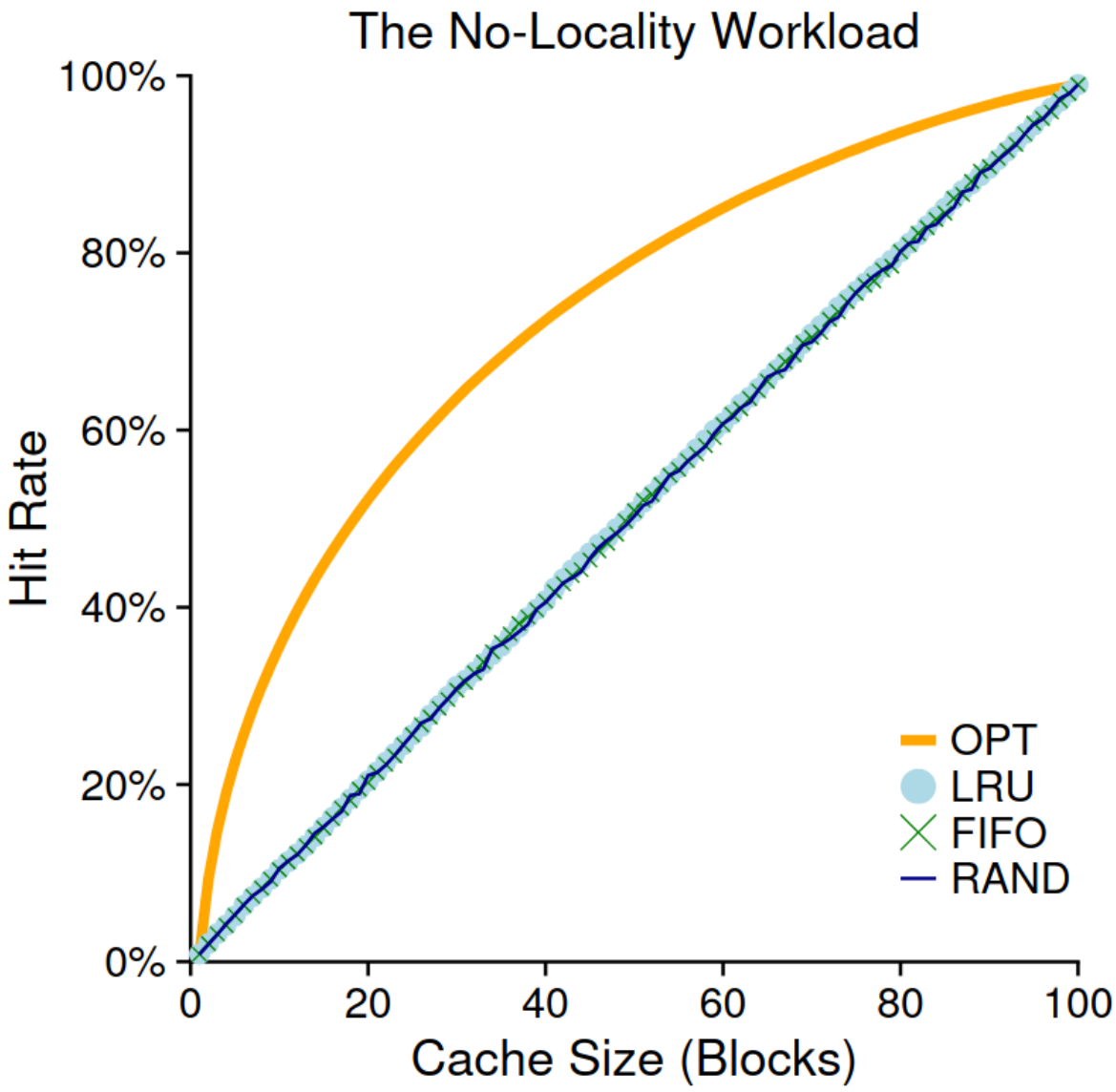


Figure 22.6: **The No-Locality Workload**

Workload with no locality is hard to predict.

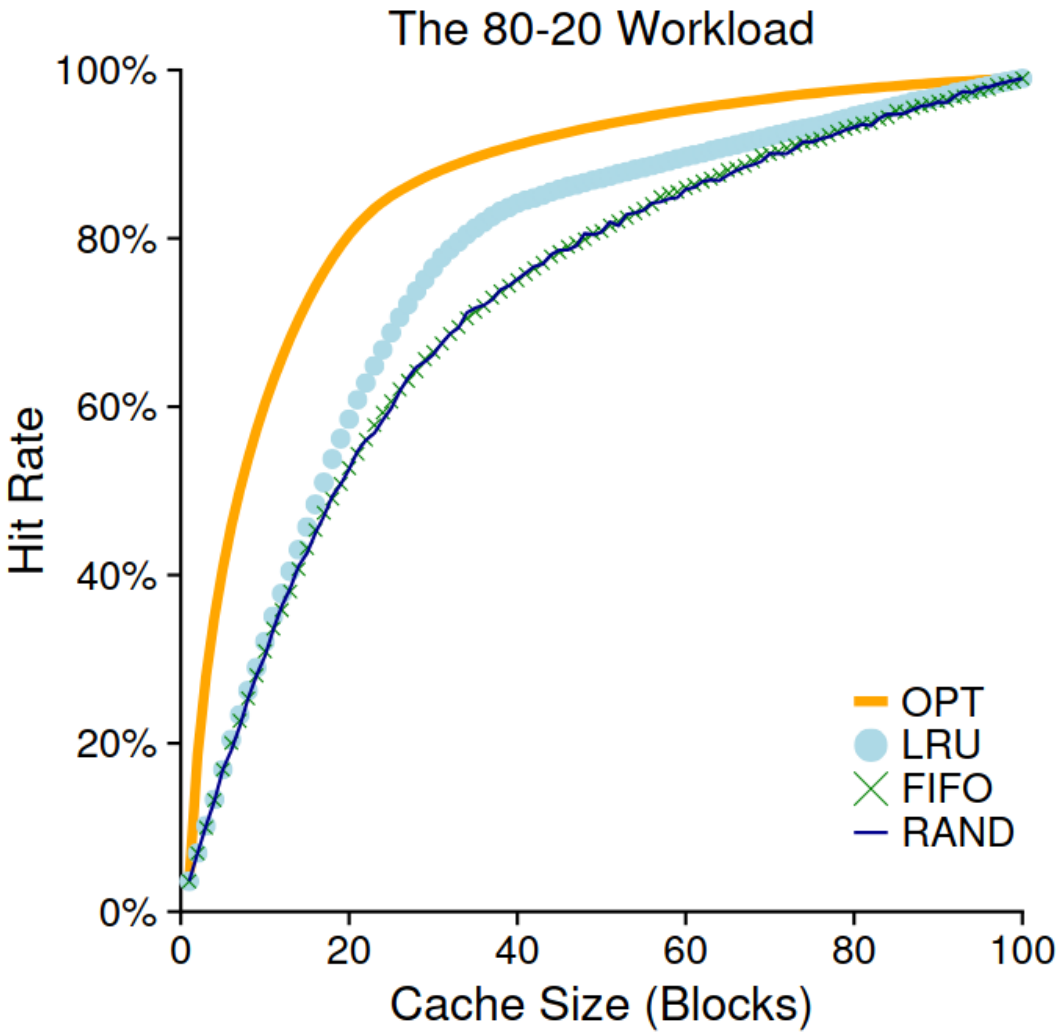


Figure 22.7: **The 80-20 Workload**

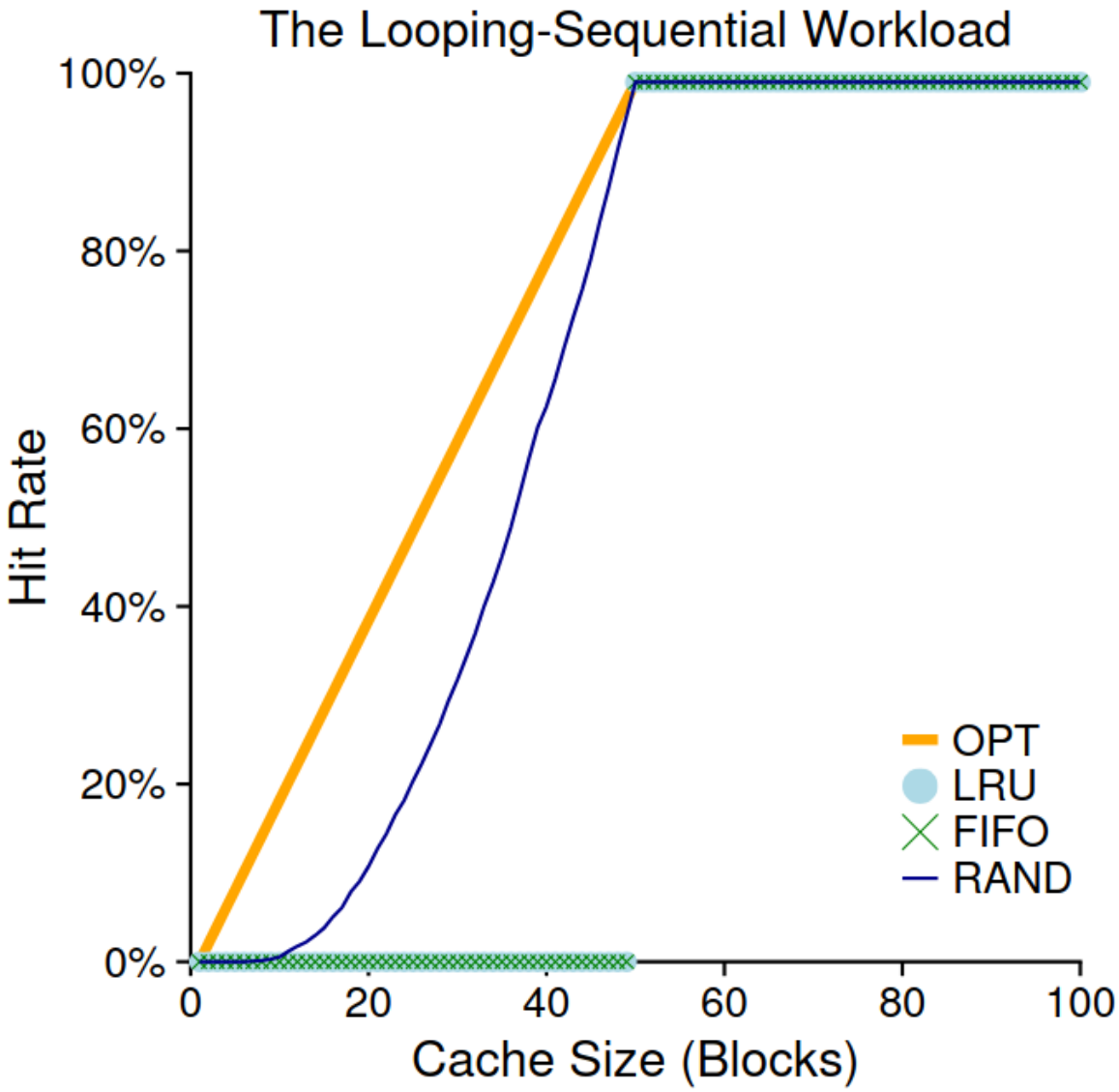


Figure 22.8: **The Looping Workload**

6.4.5 Clock

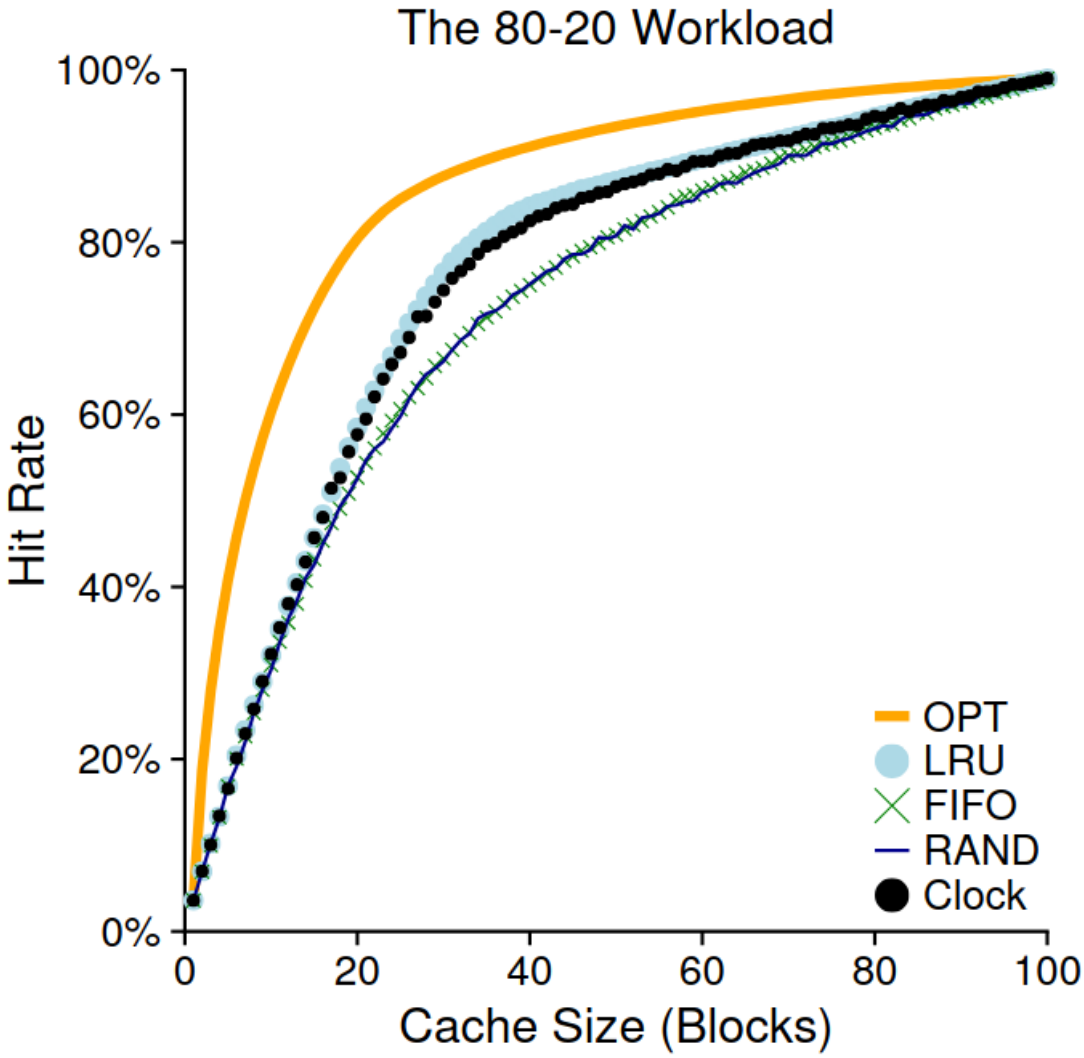


Figure 22.9: **The 80-20 Workload With Clock**

Clock is a simple approximation of LRU. It is not as good as LRU but is much easier to implement. Algorithm:

- When a page is accessed, set the reference bit to 1 (if it is not already 1)
- When a page is replaced, if the reference bit is 1, set it to 0 and move on to the next page (in a circular fashion)
- If the reference bit is 0, replace the page and set the reference bit to 1

6.4.6 Other Policies

Demand Paging: Only bring in pages when they are needed. **pre-paging**: Bring in pages before they are needed (the OS tries to predict the future).

6.5 Thrashing

> When a system spends most of its time swapping pages in and out of memory, rather than executing code.

Working set: The set of pages that a process is actively using.

6.6 Linux VM

kernel logical memory cannot be swapped to disk

Review questions and problems

1. In memory management, what do we mean with working set and thrashing?
    - By working set we refer to the pages that are actively being used by a process.
    - *Thrashing* When a system spends most of its time swapping pages in and out of memory, rather than executing code.
2. Which methods can we use to reduce the size of a pagetable in memory?
    - Multi-level page tables
3. Affinity scheduling (”CPU pinning”) decreases the number cache misses. Does it also decrease the number of TLB misses? Does it also decrease the number of page faults? Justify your answer
    - Aaffinity scheduling is about placing a thread on a CPU where it was running previous. It will also be able to reduce the number of TLB misses since TLB is present each CPU, but that doesn't help against page faults since the memory is shared between them all the CPUs. That is, if a page is in memory for one CPU, it is in memory for all CPUs.
4. (KEY PROBLEM) Calculate the size of the bitmap in a page-based memory system with page size 4KB and physical memory of 512MB?
    - 512*2^20 / 4*2^10 = 131,072 bits = 16,384 bytes
5. (KEY PROBLEM) Assume 32-bit logical/virtual addresses, page size 4KB and two-level page table. Here are the first ten entries (and the last one) in the top-level table and in one of the second-level tables. The main part of each entry has been replaced by upper-case letters in the top-level table and lower-case letters in the second-level table.

Top-level			Second-level		
1023	-	0	1023	g	1
.			.		
.			.		
.			.		
10	-	0	10	-	0
9	A	1	9	-	0
8	E	1	8	s	1
7	-	0	7	-	0
6	-	0	6	b	1
5	-	0	5	c	1
4	P	1	4	r	1
3	-	0	3	k	1

1

2	C	1	2	-	0
1	F	1	1	-	0
0	M	1	0	a	1

1. What is hidden behind the upper-case letter in the top-level table?

    - PFN of second level table
2. What is hidden behind the lower-case letter in the second-level table?
    - The PFN of the page in physical memory
3. What do you think is the meaning of the bits in the second column of each table?
    - Present, if entry is absent or not
4. Explain how the logical/virtual address `0000 0010 0100 0000 0110 1101 1011 1010` is translated to a physical address.