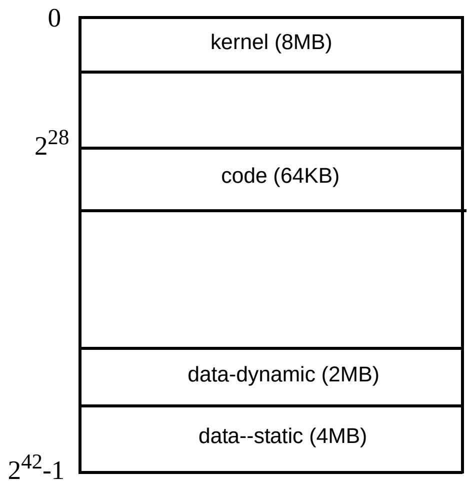
search all the sets that could contain the block on every miss. How many sets should be searched?
Number of sets: ___8_____
5. Another solution to solve the synonym problem is page coloring. What are the bits defining the color of the page?
Color bits: bits V_14_ to V_12_

Problem 4.3

This problem is about the structure of page tables to support large virtual address spaces. Assume a 42-bit virtual address space per process in a 64-bit machine and 512MByte of main memory. The page size is 4KBytes. Page table entries are 4 byte in every table. Various hierarchical page table organizations are envisioned: 1, 2, and 3 levels. The virtual space to map is populated as shown in Figure 2. Kernel space addresses are not translated because physical addresses are identical to virtual addresses. However virtual addresses in all other segments must be translated.
Please answer the following questions:

1. What would be the size of a single-level page table?
The total number of pages is 2^{30} and each entry has 4 bytes. So the page table size is 4GB. This would take more than the entire memory in most PCs today.

2. Assume now a 2-level page table. We split the 30 bits of virtual page number into two fields of 15-bits each? How many page tables would we have? What would be their total size?
The first level (root) page table must be allocated. It is accessed with 15 bits and therefore has 32K entries of 4 bytes each for a total of 128KB. Each entry of the first-level page table covers 2^{15+12} = 2^{27}=128MB of virtual memory. We don't need a second-level table for the kernel since it is not mapped. But we need a second-level page table for the code segment and the two data segments. The code segment lies in the virtual memory area covered by the second entry of the first-level page table. The two data segments lie in the virtual memory area covered by the last entry of the first-level page table. Therefore we need a total of two second-level page tables which occupy 128KB each. Thus the number of page tables is 3 and the total physical memory occupied by the page tables is 3x128KB = 384KB, which is a huge reduction as compared to the single-level page table.



3. Repeat 2. for a 3-level page table splitting the 30 bits of virtual page number into 3 fields of 10 bits each.
The root page table has 1K entries of 4 bytes each, for a total of 4KB. The kernel is unmapped. each entry of the first-level page table covers 2^{20+12}=2^{32}=4GB of virtual space. The code segment falls

within this space. So we need one second-level page table to cover the code (size 4KB). The two data segments lie in the virtual memory area covered by the last entry of the root page table. So we need another second level page table for the data (size 4KB). Each entry of the second-level page tables covers $2^{10+12} = 2^{22}$ = 4MB of virtual memory. This is enough to cover the code segment (1 third-level table or 4KB). However the data segments requires two third-level page tables as they occupy 6MB of virtual memory. The size of these two tables is 8KB. Thus the grand total is 1+2+3 = 6 tables or a total physical memory occupation of 24KB.

We see that, in terms of space (physical memory resources), having more levels in the page table hierarchy pays huge dividends. However the more levels in the hierarchy, the more time it takes to walk through the tables to translate a virtual address. This is a classical space-time trade-off. However the trade-off is very favorable in this context because a TLB can effectively bypass the whole table hierarchy.

Problem 4.4

Pseudo-LRU is an approximate LRU algorithm which requires much less overhead to manage. Thus pseudo-LRU may be less effective and cause more misses than LRU but the update of the bits required to identify the victim block is much less complex.

LRU is easily manageable for a 2-way cache. One single bit per set is sufficient to point to the LRU line. However, for larger set sizes, the complexity of LRU grows exponentially. For a 4-way cache, exact LRU needs four two bits fields, which must be updated on each access. Each two bit field is associated with a line and indicates the relative recency of access to each line in the set. This is a total of 8 bits. For a 16-way cache, exact LRU will require 16 sets of 4 bits each or 64bits total. In general, for an N-way cache, exact LRU needs $N\log_2 N$ bits.

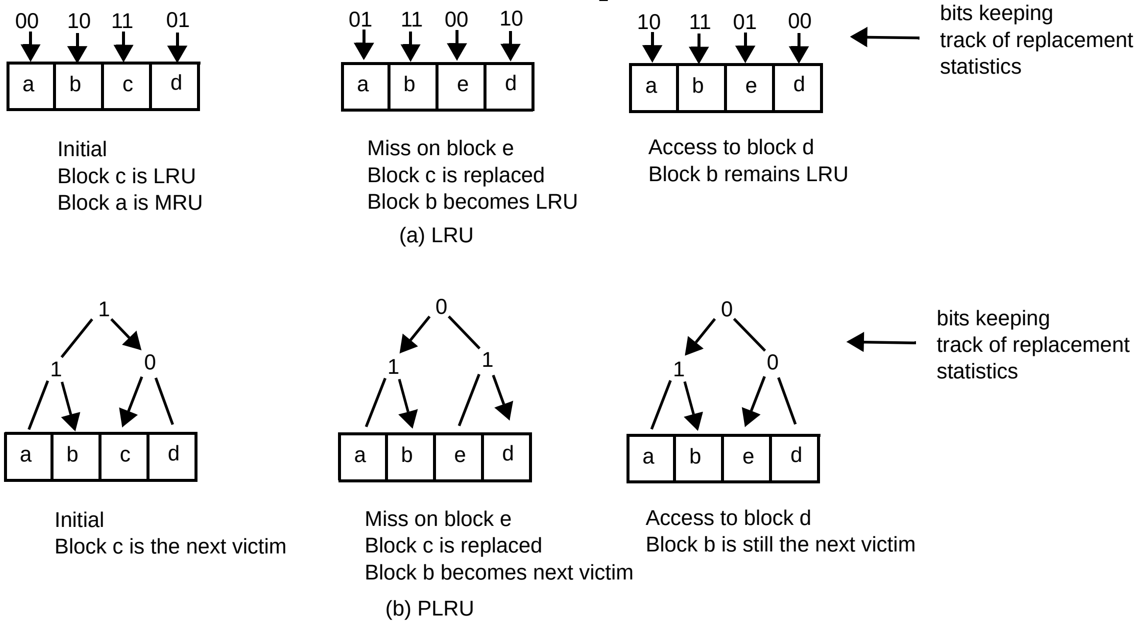


Figure 3. LRU vs. PLRU

For this reason pseudo-LRU is often preferred. Pseudo-LRU only works with power-of-two set sizes. In pseudo-LRU, the lines in each set are recursively partitioned in two subsets, and one bit points to the LRU subsets at each level of the recursive partition.

Problem 4.8

a.

Each block contains 16 words of 32 bits (4 bytes) each. There is no block re-use in the code. All misses are cold misses and therefore the cache size is irrelevant. Let's look at the time taken by a loop iteration with a load hit and a loop iteration with a load miss.

(i) cache hit:

```
LOOP:   LW R4,0(R3)          (1)
        ADDI R3,R3,stridex4  (1)
        ADD R1,R1,R4         (1)
        BNE R3,R5,LOOP       (1+2flushes)
```

(ii) cache miss:

```
LOOP:   LW R4,0(R3)          (1)
        ADDI R3,R3,stridex4  (1)
        ADD R1,R1,R4         (1+ 30 cache-latency + 1 retry)
        BNE R3,R5,LOOP       (1+2flushes)
```

The times taken by an iteration with a hit or a miss are 6 or 37 (30+1+6) respectively:

If the stride is greater than or equal to 16, every load incurs a cache miss and the average execution time per iteration is 37 cycles.

For every stride less than 16, the miss/hit sequence in a string of references is cyclic. A cycle is the minimum forward distance in the reference string so that a word with the same block offset is accessed again in the blocks that are touched. Since LCM(16,stride) is the amount of memory spanned between two consecutive accesses to the same word in blocks (LCM is the least common multiple) the number of references in a cycle is LCM(16, stride)/stride. Since the stride is less than 16, all blocks accessed in a cycle are contiguous. Thus, the number of misses in each cycle is LCM(16, stride)/16.

For example, if the stride is 1, the length of miss/hit sequence cycles is 16 (or LCM(16,1)/1), and only 1 cache miss occurs (LCM(16, 1)/16) in each cycle. The number of accesses is also the number of loop iterations, i.e., 16. So the average execution time per iteration i when the stride is 1 is (37*1 +6*15)/16 = 7.9375 cycles/iteration.

The average execution time for each loop as a function of the stride is as follows:

i) stride >= 16:

The cache misses on every iteration. The average iteration execution time per iteration is 37 cycles.

ii) stride < 16:

average iteration execution time = $\frac{(Nbofmisses \times 37) + (Nbofhits \times 6)}{Nbofaccesses}$ =

$$\frac{((LCM(16, stride)/16) \times 37) + ((LCM(16, stride)/stride - LCM(16, stride)/16) \times 6)}{(LCM(16, stride)/stride)} =$$

$$\frac{37 \times stride}{16} + 6 - \frac{6 \times stride}{16} = \left(\frac{31}{16} \times stride\right) + 6$$

To obtain this result more directly, observe that, as the length of the reference string *N* grows to infinity, the number of contiguous blocks accessed (i.e. the number of misses) tends to *N*x*stride*/16 on the average. The miss rate therefore tends to *stride*/16. The number of clocks added to an iteration that misses is 31. Thus the execution time is: $6 + \frac{stride}{16} \times 31$.

b.

The number of cycles taken by a loop iteration with a cache hit increases to 7 because the PW instruction consumes one additional cycle.

Because the prefetch instruction is only one iteration ahead of the load instruction and the stride of the prefetch instruction is the same as the stride of the load instruction, the prefetch instruction is beneficial when the prefetch instruction accesses the next cache block. It cuts the penalty of a cache miss by the total number of cycles between the prefetch instruction and the load instruction of the next iteration (i.e., 6 cycles), so that the miss cost is 26 cycles (=32-6). Because the prefetch consumes one cycle, the time taken by an iteration with a miss is 37 + 1 -6 = 32. We just plug this new penalty number into the equation obtained in part a.

i) stride >= 16

As in (a) above, every load incurs a cache miss. However, now, because of the prefetch, the miss latency of the load is reduced to 26 cycles.

average iteration execution time = 32 cycles

ii) stride < 16

average iteration execution time = $\dfrac{(Nbofmisses \times 32) + (Nbofhits \times 7)}{Nbofaccesses}$

$$= \frac{((LCM(16, stride)/16) \times 32) + (((LCM(16, stride)/stride) - (LCM(16, stride)/16)) \times 7)}{(LCM(16, stride)/stride)}$$

$$= \left(\frac{25}{16} \times stride\right) + 7$$

c.

i) For strides that are greater than or equal to 16, prefetch is always more effective as the execution time of every iteration is 32 cycles, which is less than without prefetch (37 cycles.)

ii) stride < 16

For prefetch to be effective we must have:

$$\left(\frac{25}{16} \times stride\right) + 7 < \left(\frac{31}{16} \times stride\right) + 6$$

$$1 < \left(\frac{6}{16} \times stride\right)$$

$$\frac{16}{6} < stride$$

$$3 \le stride$$

Let’s say that we have N lines in a set, where N=2^n. These N lines are split into two subsets of size N/2 each. One bit is enough to track the LRU subset. Then each subset is again split into two sub-subsets of size N/4 each, which requires two bits to track the LRU sub-subsets in the subsets, for a total of 3 bits. Next we split the sub-subsets again into sub-sub-subsets of size N/8 each, which requires four bits, for a total of 7 bits. We do this recursively until we cannot split anymore because we have reached individual lines. The total number of bits is $\log_2 N$-1, much less than for LRU. For example, for a 16-way cache, the number of bits is 15, as compared to 64 for LRU. Therefore the FSM updating the replacement policy bits is much simpler.

When the set size is 2, LRU and pseudo-LRU are identical. Only 1 bit is needed. When it is more than 2 lines, the bit at each level points to the LRU subset at the next level. So we keep track of the LRU subset at each level.

The difference between LRU bits and pseudo-LRU (PLRU) bits management is illustrated in Figure 3 for a set size of 4 cache lines (to simplify, the cache size is four lines).

The four lines are divided into two subsets: S0={line0,line1} and S1={line2,line3}. Each subset is subdivided into two lines. In the initial state, S0 contains blocks a and b and S1 contains blocks c and d. At the root of the tree bit value 0 points to S0 and bit value 1 points to S1. This correspondence is indicated by the arrows. Within S0, one bit points to line0 (value 0) or line1 (value 1). A bit always points away from the MRU subset and towards the LRU subset. So in the initial state, the MRU subset is S0 and the LRU subset is S1. Within S1, the LRU line is line2 (containing block c) and the MRU line is line 3. This is consistent with the initial state of LRU. After a miss on block e the victim is pointed to by the arrows, i.e., block c contained in line2. At this point the LRU subset becomes S0 and line1 is the new candidate for replacement. Then block d is accessed. S1 remains the MRU subset and line1 contains the next victim, i.e., block b. This is still consistent with LRU.

PLRU approximates LRU, and in this example it points to the same victim as LRU. However, PLRU will eventually diverge from LRU.

Problem 4.5

a.
DIRECT-MAPPED:

Table 48: Direct-Mapped

Cycle #	1						2						3						4					
Block Address	0	1	2	3	4	5	0	1	2	3	4	5	0	1	2	3	4	5	0	1	2	3	4	5
Line0	0	0	0	0	4	4	0	0	0	0	4	4	0	0	0	0	4	4	0	0	0	0	4	4
Line1	-	1	1	1	1	5	5	1	1	1	1	5	5	1	1	1	1	5	5	1	1	1	1	5
Line2	-	-	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
Line3	-	-	-	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
Misses	*	*	*	*	*	*	*	*			*	*	*	*			*	*	*	*			*	*

In the first cycle, all blocks (0,1,2,3,4,5) miss in the cache (cold misses). From then on the blocks mapping to lines 0 and 1 keep missing (4 misses) while blocks 2 and 3 remain in lines 2 and 3 (2 hits) in every cycle. So, after 6 misses in the first cycle, all subsequent cycles experience 4 misses.

Total number of misses = 6 + (4 * 9) = **42 misses**

FA with LRU REPLACEMENT:

Table 49: FA with LRU

Cycle #	**1**						**2**						**3**						**4**					
Block Address	0	1	2	3	4	5	0	1	2	3	4	5	0	1	2	3	4	5	0	1	2	3	4	5
Line0	0	0	0	0	4	4	4	4	2	2	2	2	0	0	0	0	4	4	4	4	2	2	2	2
Line1	-	1	1	1	1	5	5	5	5	3	3	3	3	1	1	1	1	5	5	5	5	3	3	3
Line2	-	-	2	2	2	2	0	0	0	0	4	4	4	4	2	2	2	2	0	0	0	0	4	4
Line3	-	-	-	3	3	3	3	1	1	1	1	5	5	5	5	3	3	3	3	1	1	1	1	5
Misses	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*

In the first iteration, all blocks (0,1,2,3,4,5) miss in the cache (cold misses). From then on the cache misses at every reference because the cache cannot hold block copies across cycles. LRU copies are replaced and then accessed again. Thus we have 6 misses per cycle.

Total number of misses = 6 * 10 = **60 misses**

FA with FIFO REPLACEMENT:

Table 50: FA with FIFO

Cycle #	**1**						**2**						**3**						**4**					
Block Address	0	1	2	3	4	5	0	1	2	3	4	5	0	1	2	3	4	5	0	1	2	3	4	5
Line0	0	0	0	0	4	4	4	4	2	2	2	2	0	0	0	0	4	4	4	4	2	2	2	2
Line1	-	1	1	1	1	5	5	5	5	3	3	3	3	1	1	1	1	5	5	5	5	3	3	3
Line2	-	-	2	2	2	2	0	0	0	0	4	4	4	4	2	2	2	2	0	0	0	0	4	4
Line3	-	-	-	3	3	3	3	1	1	1	1	5	5	5	5	3	3	3	3	1	1	1	1	5
Misses	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*

In the first iteration, all blocks (0,1,2,3,4,5) miss in the cache (cold misses). From then on, the cache misses at every reference. Because the reference string is cyclic, the FIFO cache behaves like the LRU cache. Again we have 6 misses per cycle.

Total number of misses = 6 * 10 = **60 misses**
FA with LIFO REPLACEMENT:

Table 51: FA with LIFO

Cycle #	**1**						**2**						**3**						**4**					
Block Address	0	1	2	3	4	5	0	1	2	3	4	5	0	1	2	3	4	5	0	1	2	3	4	5

Table 51: FA with LIFO

Cycle #	1						2						3						4					
Line0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Line1	-	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Line2	-	-	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
Line3	-	-	-	3	4	5	5	5	5	3	4	5	5	5	5	3	4	5	5	5	5	3	4	5
Misses	*	*	*	*	*	*				*	*	*				*	*	*				*	*	*

In the first iteration, all blocks (0,1,2,3,4,5) miss in the cache (cold misses). From then on, blocks 0,1,and 2 stay in cache and the block in line 3 is constantly victimized to hold lines 3, 4 and 5. So, we have 3 misses per cycle.

Total number of misses = 6 + (3 * 9) = **33 misses**

2-way SA with LRU REPLACEMENT:

Table 52: 2-way SA with LRU per set

Cycle #	1						2						3						4					
Block Address	0	1	2	3	4	5	0	1	2	3	4	5	0	1	2	3	4	5	0	1	2	3	4	5
Line0	0	0	0	0	4	4	4	4	2	2	2	2	0	0	0	0	4	4	4	4	2	2	2	2
Line1	-	-	2	2	2	2	0	0	0	0	4	4	4	4	2	2	2	2	0	0	0	0	4	4
Line2	-	1	1	1	1	5	5	5	5	3	3	3	3	1	1	1	1	5	5	5	5	3	3	3
Line3	-	-	-	3	3	3	3	1	1	1	1	5	5	5	5	3	3	3	3	1	1	1	1	5
Misses	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*

Set 0 is made of Line 0 and 1 and set 1 is made of Line 2 and 3. We can see the reference string of 5 as two interleaved strings one with odd numbered block and one with even numbered block. Each of the interleaved string access an LRU cache of size 2. As with the case for the FA LRU cache, since each string length (3) is greater than the cache size (2), every access misses. So we have 6 misses per cycle.

Total number of misses = 6 * 10 = **60 misses.**

The interesting observation from all this is that, in cyclic reference patterns (such as to instructions in loops), the FA-LIFO policy is the best, followed by direct-mapped mapping.

b.
COLD MISSES:
The number of cold misses is independent of the cache organization and replacement policy. The number of cold misses is equal to the number of blocks in the trace:

Nb_cold_misses = **6 misses**

CAPACITY MISSES:
The number of capacity misses is also independent of the cache organization and replacement policy. It is given by the number of misses in an FA LRU cache minus the number of cold misses:

Nb_capacity_misses = 60 - 6 = **54 misses**

CONFLICT MISSES:
The number of conflict misses depends on the cache organization and replacement policy. It is obtained by subtracting both cold and capacity misses from the total number of misses of the cache:

i) Direct-mapped: Nb_conflict_misses = 42 - (6 + 54) = **-18 misses**

ii) FA with LRU: Nb_conflict_misses = 60 - (6 + 54) = **0**

iii) FA with FIFO: Nb_conflict_misses = 60 - (6 + 54) = **0**

iv) FA with LIFO: Nb_conflict_misses = 33 - (6 + 54) = **-27 misses**

v) 2-way SA with LRU for each set: Nb_conflict_misses = 60 - (6 + 54) = **0**

Since we are using a FA cache with LRU to calculate the capacity misses, we end up having negative number of conflict misses for some cache organization and this of course is not acceptable. The problem is FA cache with LRU is not the best cache and hence can not be used, instead we should use a FA cache with optimal replacement algorithm as we will do in part c and part d.

c.
FA with OPT REPLACEMENT:
The cache states for the entire sequence of references is shown in Table 53. After the cache is filled, after access #4, two accesses misses and refill the LIFO position of the stack, then three accesses hit on the other three line. This pattern with cycle 5 repeats until the end of the trace. Because the number of remaining accesses after 4 is not a multiple of 5, the last cycle is truncated.
Total number of misses = 4 + (59-4)/5*2 + 1 = 4+22+1 = **27 misses**

d.
Conflict misses are counted for each cache by subtracting the number of misses in FA-OPT from the number of misses in each cache:

i) Direct-mapped: Nb_conflict_misses = 42 - 27 = **15 misses**

ii) FA with LRU: Nb_conflict_misses = 60 - 27 = **33 misses**

iii) FA with FIFO: Nb_conflict_misses = 60 - 27 = **33 misses**

iv) FA with LIFO: Nb_conflict_misses = 33 - 27 = **6 misses**

v) 2-way SA with LRU for each set: Nr_conflict_misses = 60 - 27 = **33 misses**
By using a FA cache with OPT replacement policy as the baseline to count capacity misses, the number of conflict misses is positive for all cache organizations. This is clearly the appropriate way to classify misses. However, the number of misses in FA-LRU is easier to count and FA-LRU may

be acceptable as a baseline for all practical purposes.

Table 53: FA with OPT

Cycle #	1						2						3						4					
Block Address	0	1	2	3	4	5	0	1	2	3	4	5	0	1	2	3	4	5	0	1	2	3	4	5
Line0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	2	2	2	2
Line1	-	1	1	1	1	1	1	1	1	1	1	1	1	1	2	3	3	3	3	3	3	3	3	3
Line2	-	-	2	2	2	2	2	2	2	3	4	4	4	4	4	4	4	4	4	4	4	4	4	4
Line3	-	-	-	3	4	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5
Misses	*	*	*	*	*	*				*	*				*	*				*	*			
Cycle #	5						6						7						8					
Block Address	0	1	2	3	4	5	0	1	2	3	4	5	0	1	2	3	4	5	0	1	2	3	4	5
Line0	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	3	4	4	4	4	4	4	4	4
Line1	3	3	3	3	3	3	3	3	3	3	4	5	5	5	5	5	5	5	5	5	5	5	5	5
Line2	4	4	4	4	4	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Line3	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	2	3	3	3
Misses	*	*				*	*				*	*				*	*				*	*		
Cycle #	9						10																	
Block Address	0	1	2	3	4	5	0	1	2	3	4	5												
Line0	4	4	4	4	4	4	4	4	4	4	4	5												
Line1	5	5	5	5	5	5	0	1	1	1	1	1												
Line2	0	1	2	2	2	2	2	2	2	2	2	2												
Line3	3	3	3	3	3	3	3	3	3	3	3	3												
Misses		*	*				*	*				*												

Problem 4.6

a.
i) LRU:

Table 54: LRU

trace	a	a	b	c	a	a	d	e	f	f	e	f	e	f	e	f	a	b	g	c	a	e	f
Miss/Hit	M	H	M	M	H	H	M	M	M	H	H	H	H	H	H	H	H	M	M	M	H	M	M
LRU priority list	a	a	b a	c b a	a c b	a c b	d a c b	e d a c	f e d a	f e d a	e f d a	f e d a	e f d a	f e d a	e f d a	f e d a	a f e d	b a f e	g b a f	c g b a	a c g b	e a c g	f e a c