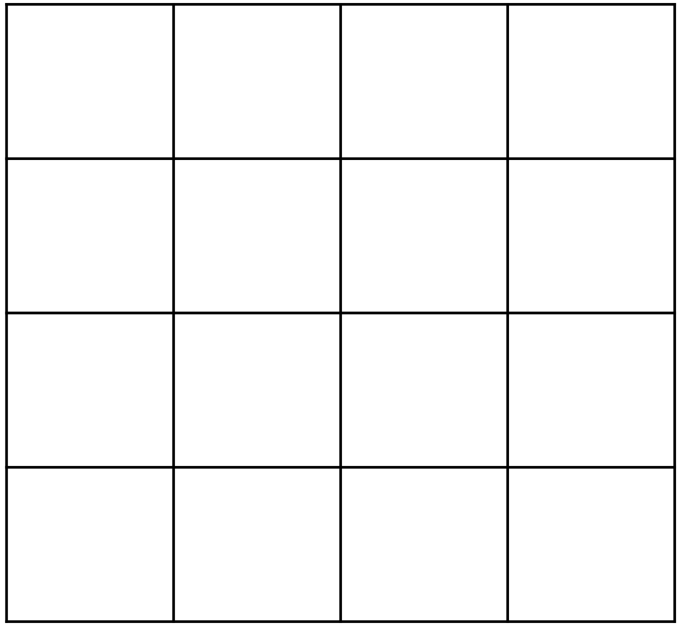
Caches and the Memory Hierarchy

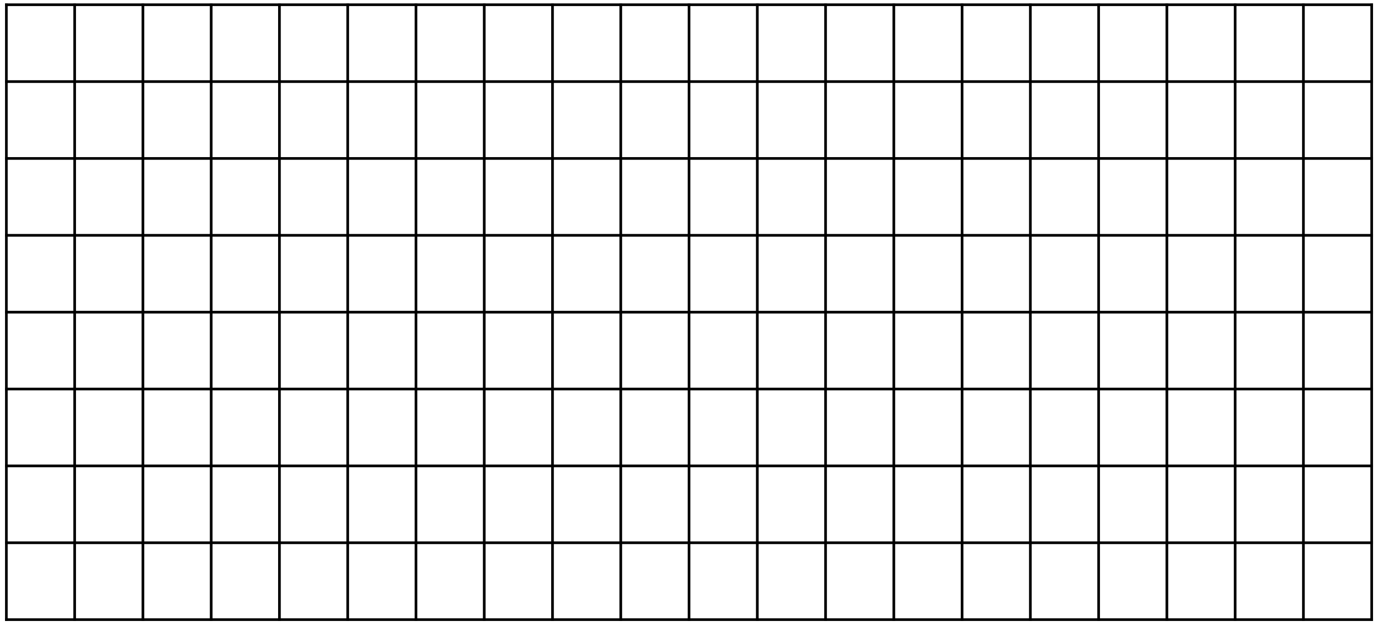
Latency, size and price of computer memory

Given a budget, we need to trade off

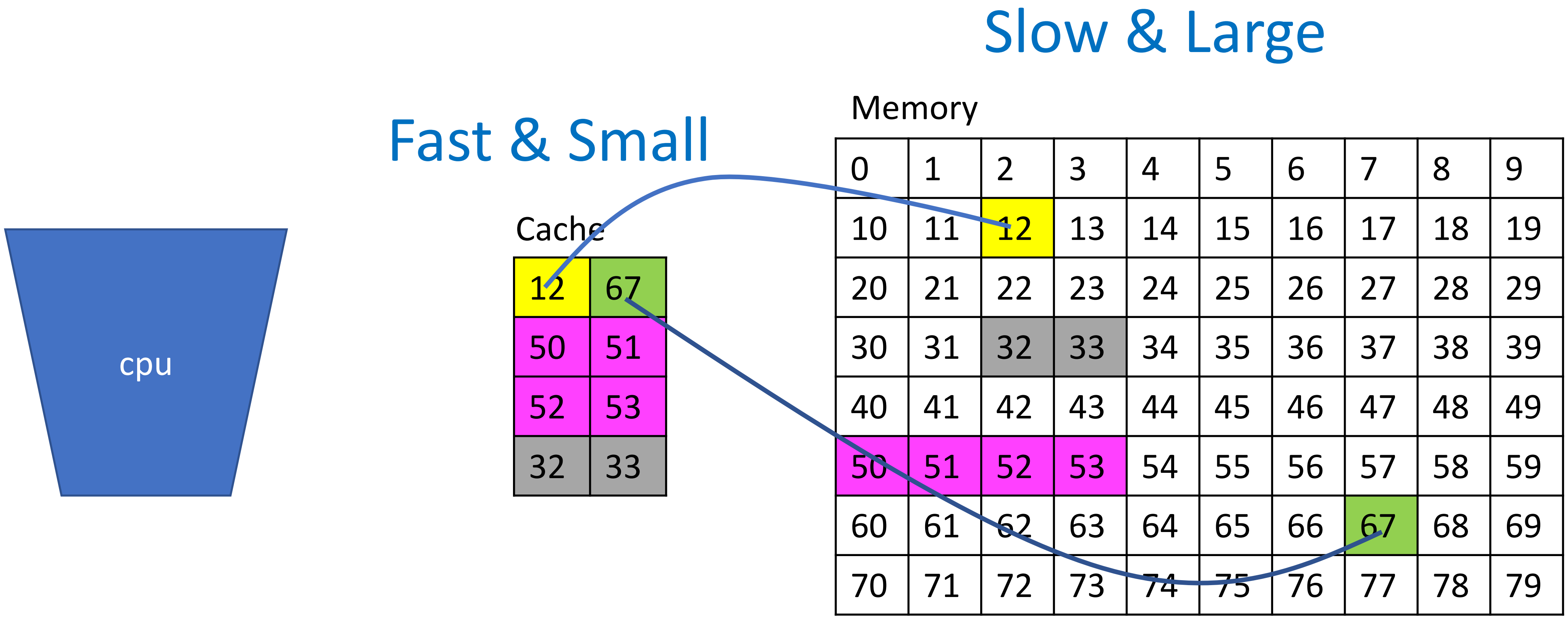
$10: Fast & Small



$10: Slow & Large



Cache: The basic idea



Slow & Large

Fast & Small

cpu

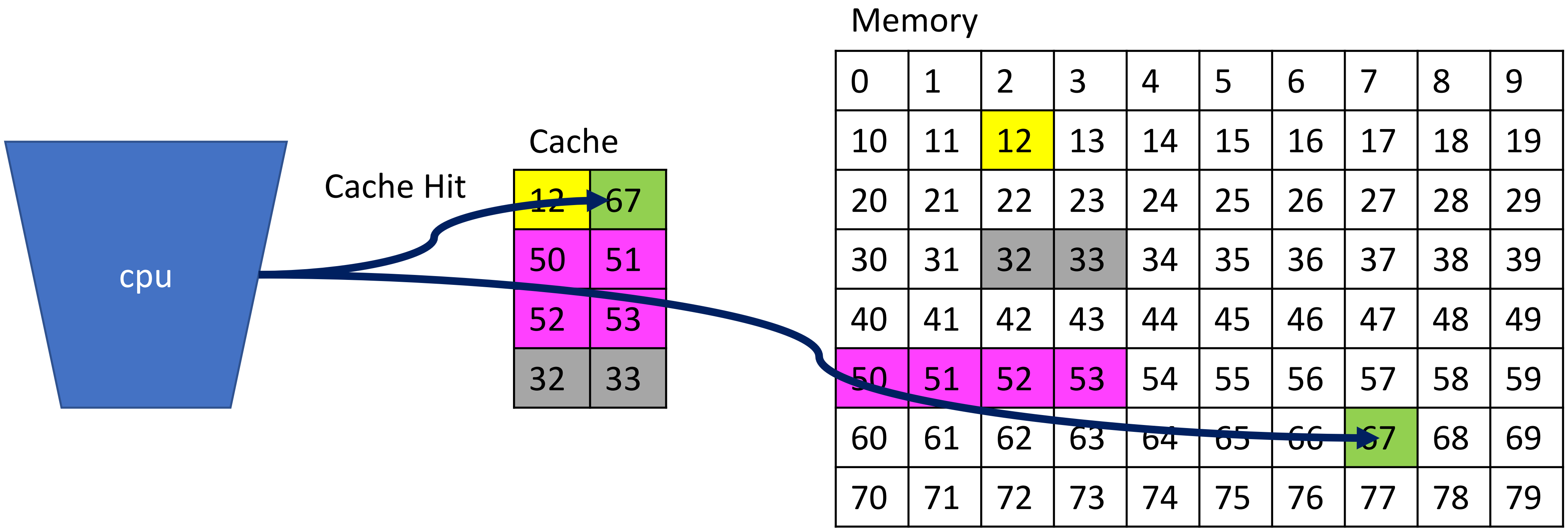
Cache

12	67
50	51
52	53
32	33

Memory

0	1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	27	28	29
30	31	32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47	48	49
50	51	52	53	54	55	56	57	58	59
60	61	62	63	64	65	66	67	68	69
70	71	72	73	74	75	76	77	78	79

Cache Hit



cpu

Cache Hit

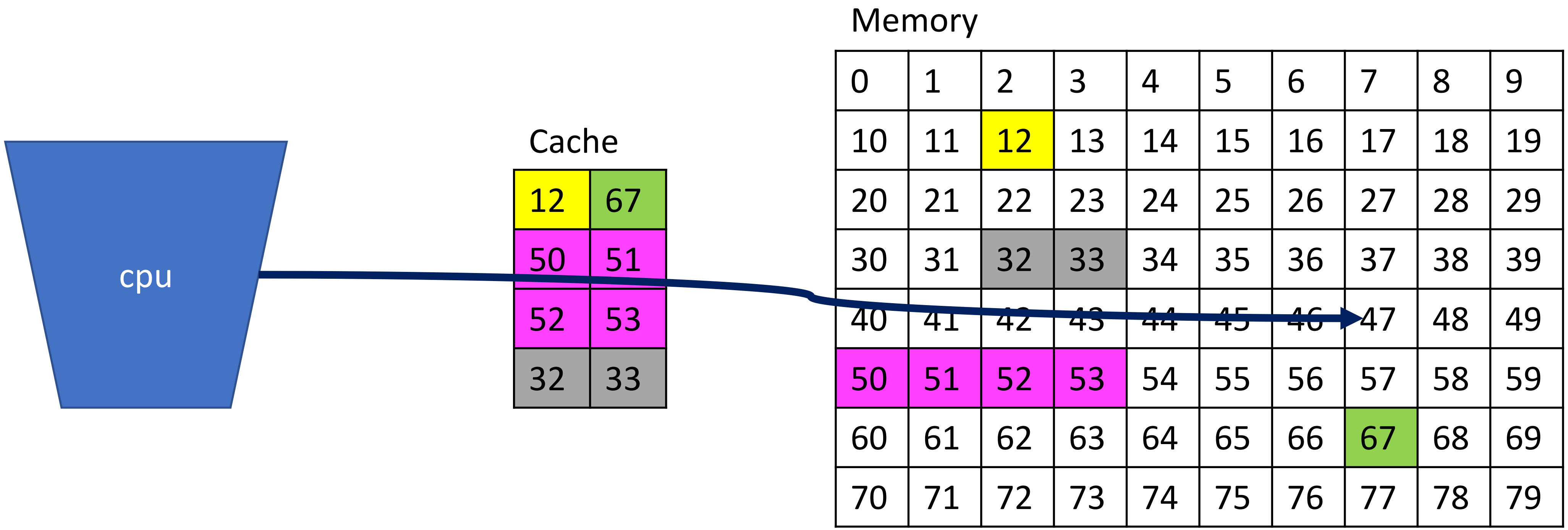
Cache

12	67
50	51
52	53
32	33

Memory

0	1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	27	28	29
30	31	32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47	48	49
50	51	52	53	54	55	56	57	58	59
60	61	62	63	64	65	66	67	68	69
70	71	72	73	74	75	76	77	78	79

Cache Miss



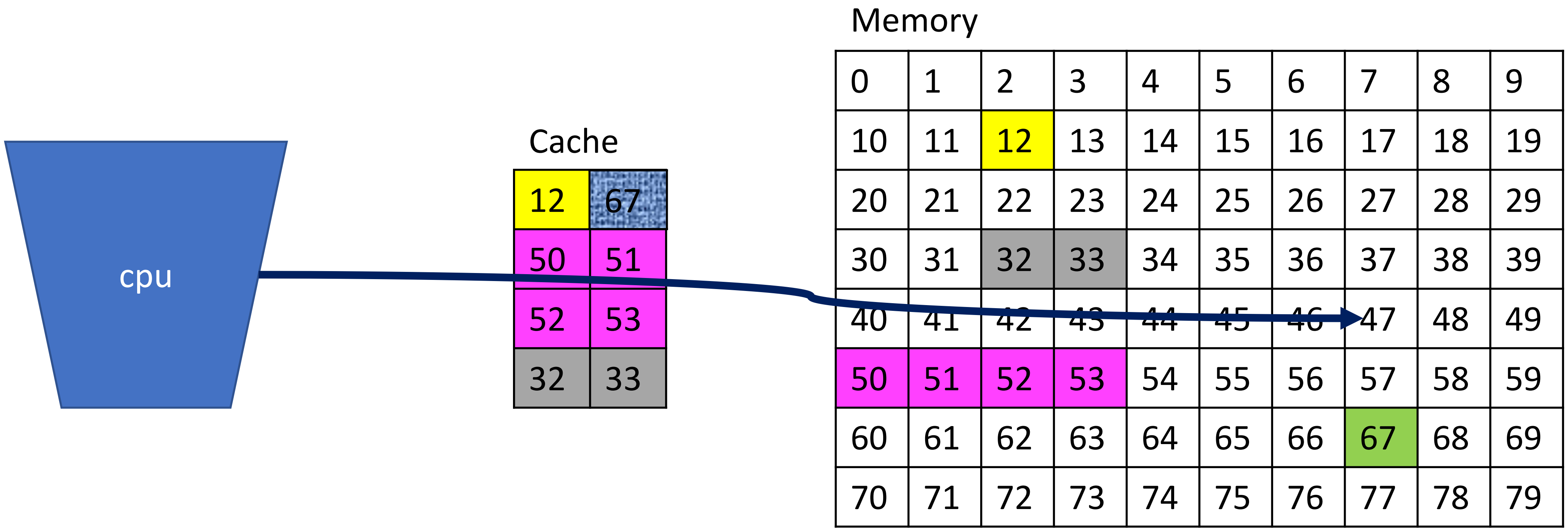
Cache

12	67
50	51
52	53
32	33

Memory

0	1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	27	28	29
30	31	32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47	48	49
50	51	52	53	54	55	56	57	58	59
60	61	62	63	64	65	66	67	68	69
70	71	72	73	74	75	76	77	78	79

Cache Miss Service: 1) Choose byte to drop



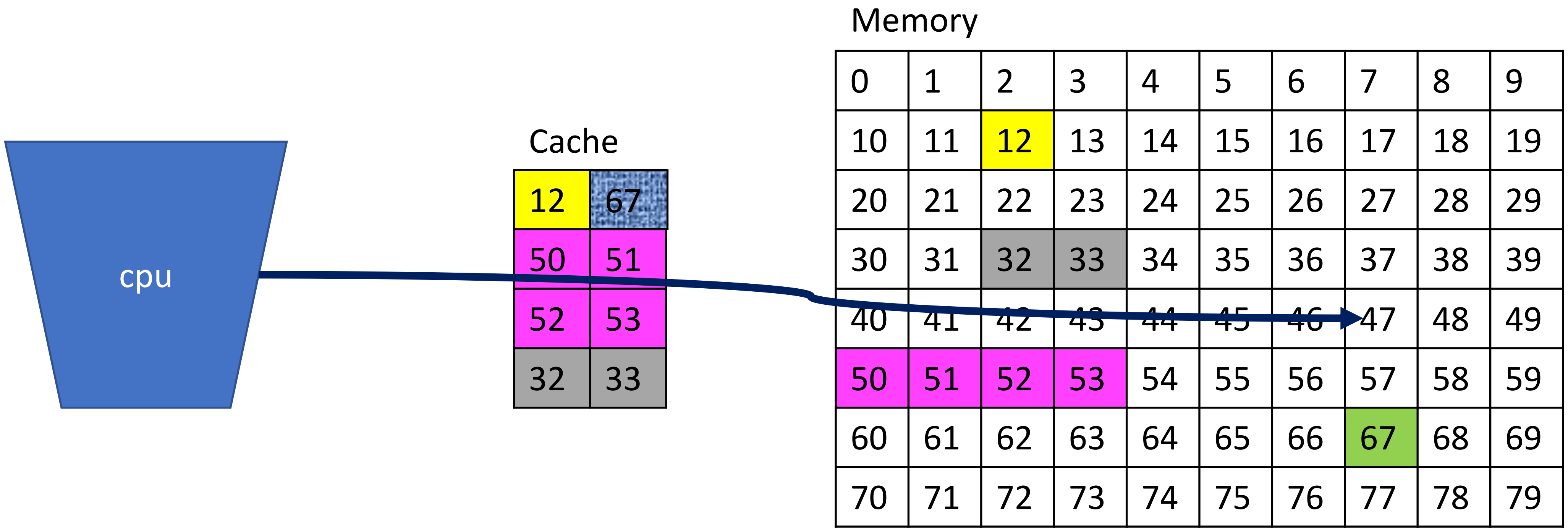
Cache

12	67
50	51
52	53
32	33

Memory

0	1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	27	28	29
30	31	32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47	48	49
50	51	52	53	54	55	56	57	58	59
60	61	62	63	64	65	66	67	68	69
70	71	72	73	74	75	76	77	78	79

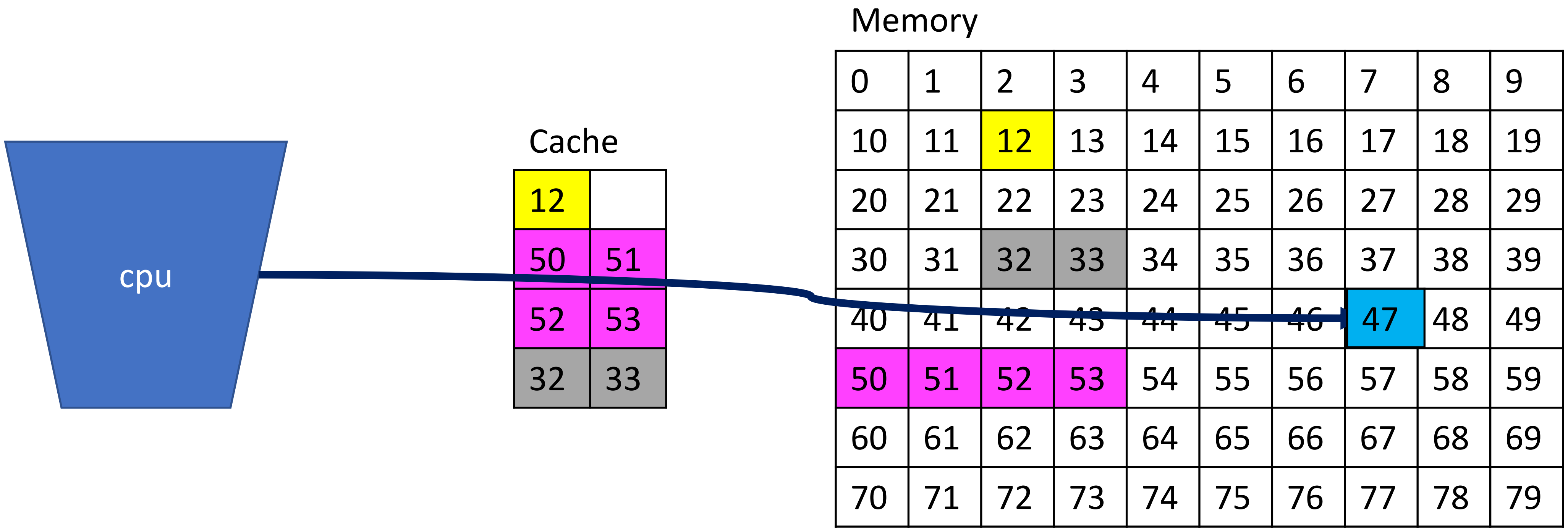
Cache Miss Service: 2) write back



12	67
50	51
52	53
32	33

0	1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	27	28	29
30	31	32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47	48	49
50	51	52	53	54	55	56	57	58	59
60	61	62	63	64	65	66	67	68	69
70	71	72	73	74	75	76	77	78	79

Cache Miss Service: 3) Read In



Cache

12	
50	51
52	53
32	33

Memory

0	1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	27	28	29
30	31	32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47	48	49
50	51	52	53	54	55	56	57	58	59
60	61	62	63	64	65	66	67	68	69
70	71	72	73	74	75	76	77	78	79

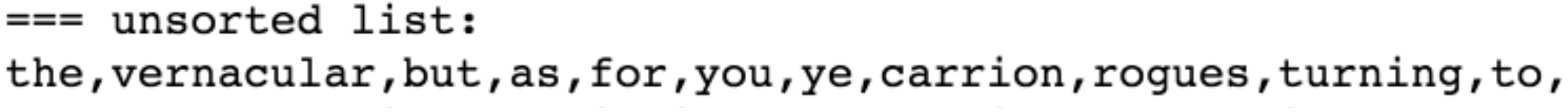
Access Locality

- The cache is effective If most accesses are hits.
  - Cache Hit Rate is high.
- **Temporal Locality**: Multiple accesses to **same** address within a short time period

Spatial locality

- **Spatial Locality**: Multiple accesses to close-together addresses in short time period.
  - The difference between two sums.
  - Counting words by sorting
- Benefiting from spatial locality
  - Memory is partitioned into **Blocks/Lines** rather than single bytes.
  - Moving a block of memory takes much less time than moving each byte individually.
  - Memory locations that are close to each other are likely to fall in the same block.
  - Resulting in more cache hits.

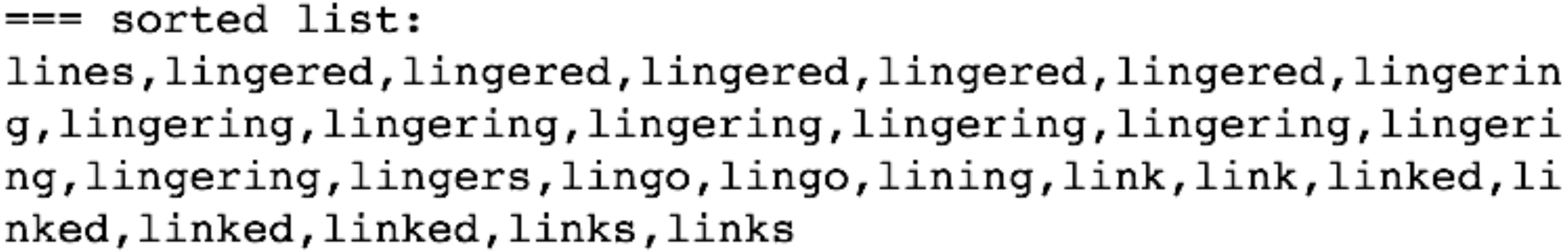
Unsorted word count / poor locality



```
=== unsorted list:
the,vernacular,but,as,for,you,ye,carrion,rogues,turning,to,
```

- Consider the memory access to the dictionary D:
- Count without sort:
  D[the]=12332,…,D[but]=943,………,D[vernacular]=10,………….,D[for]=..
- Temporal locality for very common words like “the”
- No spatial locality

sorted word count / good locality



```
=== sorted list:
lines,lingered,lingered,lingered,lingered,lingered,lingerin
g,lingering,lingering,lingering,lingering,lingering,lingeri
ng,lingering,lingers,lingo,lingo,lining,link,link,linked,li
nked,linked,linked,links,links
```

Entries to D are added one at a time.

1. D[lines]=33
2. D[lines]=33, D[lingered]=5
3. D[lines]=33, D[lingered]=5, D[lingering]=8

Assuming new entries are added at the end, this gives spatial locality.

Spatial locality makes code run much faster (X300)

Summary

- Caching reduces storage latency by bringing relevant data close to the CPU.
- This requires that code exhibits access locality:
    - Temporal locality: Accessing the same location multiple times
    - Spatial locality: Accessing neighboring locations.