LRU: Partition on demand basis

Motivation: Higher demand for cache resources does not always correlate with a higher performance from additional cache resources.

→ (Eg.) A scanning will have high demand but there will be no reuse, hence it is not useful to alot it in cache

↓

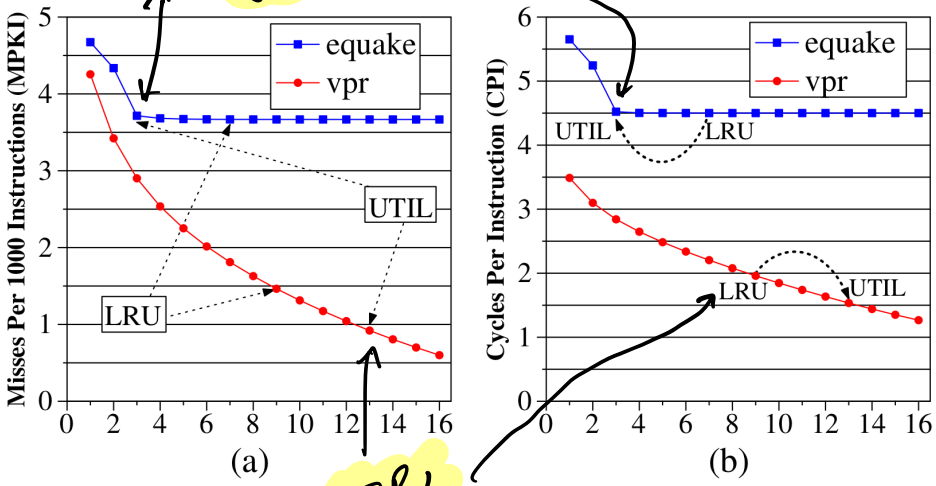
Makes sense to give more cache to appli that will reuse data

↳ UCP (Utility-based Cache Partitioning)

↳ Low-overhead runtime

↳ Based on misses from each application given an amount of cache.

Equake doesn't benefit beyond certain point



VPR benefits from increase in cache.

What we want to do?

→ Partition cache based on utility (UTIL)

↳ Utility based Cache Partitioning (UCP)

↳ UMON (Utility Monitoring)

↳ Lookahead.

Types of utility:

→ Low utility:
* Won't benefit from increasing the cache.
* But will help if increased exponentially
* Scanning / Thrashing

→ High utility
* Benefits from increasing the cache

→ Saturating utility
* Small workload that fits in small cache. So won't benefit from increasing cache.

Combination of workloads:

Workload 1	Workload 2	
Low util	Low util	No issues
Saturating util	Saturating util	No issues
Saturating util	Low util	Cache may not accomodate working set of saturating util.
High util	Saturating util	Issue. Highly sensitive to cache size
High util	Low util	

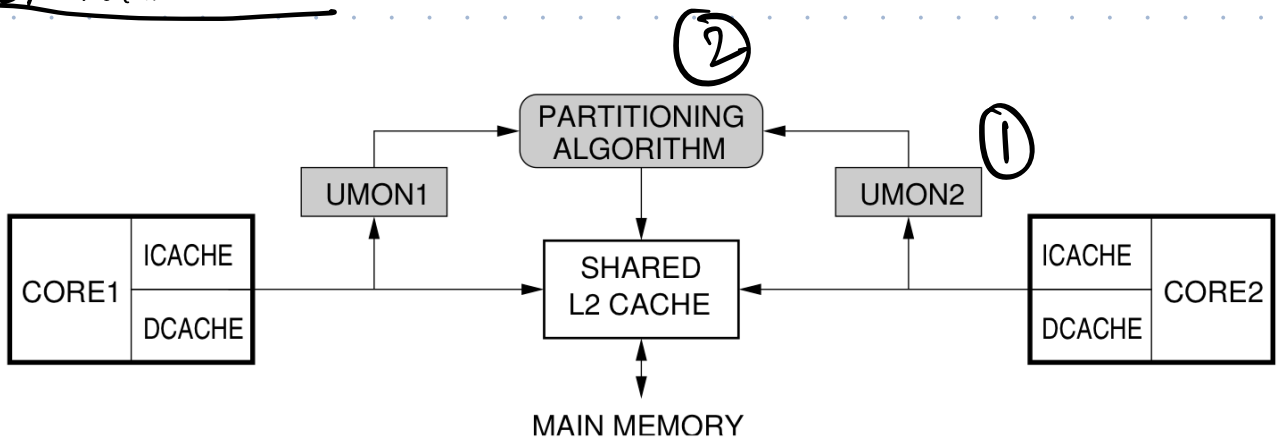
→ Reduction in MPKI ∝ Reduction in CPI

Defining utility:

→ $U_a^{da} = miss_a - miss_{da}$ (a < da)

$miss_a$: Miss occured when a ways are allocated

UCP Framework:



Totally there are 3 components:

1) UMON: (Utility Monitor)
   → Collects util information

2) Partitioning Algorithm:
   → Partitions the ways based on info from UMON

3) Replacement Engine
   → Augmented to support the partitions made by partitioning algorithm!

① UMON:

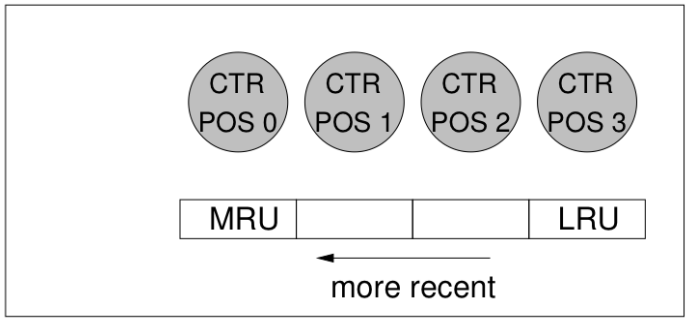
→ We only need tag directories

→ Naive but expensive way: Check misses for 1–16 ways.

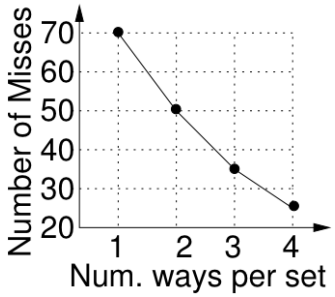
↳ Instead exploit stack property of LRU.

↳ Maintain a counter for each way. Whenever there is a hit in a way, increment the corresponding counter.

These counter values indicate the number of misses if this way didn't exist.



HIT COUNTER	Value
CTR POS 0	30
CTR POS 1	20
CTR POS 2	15
CTR POS 3	10
MISSES = 25	



To get misses for way=2, add 10+15. Take into account the cumsum.

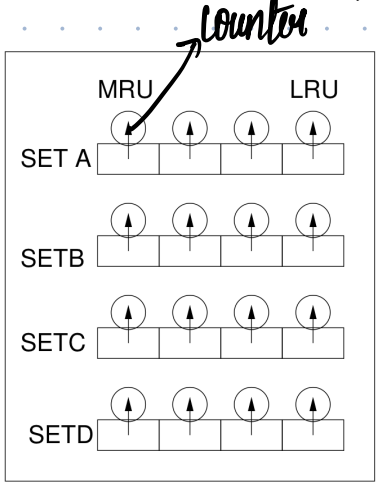
Therefore, inorder to obtain misses for all ways, we just have to do it only for 16-ways, with only tag dirs.

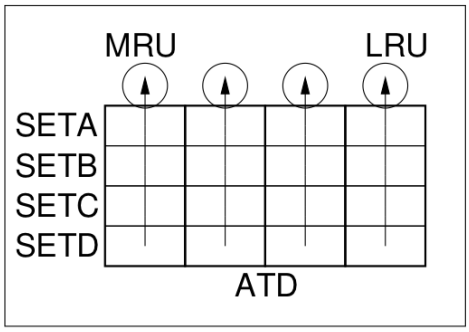
How does the circuit of UMON work?

→ Auxuillary Tag Directory (ATD): Has same associativity as the main tag array of shared cache.

→ Hit counters

* UMON - Local
  - Has counter for each block
  - Total "sets × ways" counters
  - Alot of overhead.





* UMON-global

- Has one counter per way.
- Total "ways" counter

→ Comparitively less number of counters

Issue with UMON-global:

→ has the extra overhead of maintaining ATD.

→ 1 UMON per core.

→ Eg.)

Tag = 4 bytes.

We need + (4 x cores) overhead!

Solution: UMON-DSS

= UMON-global + Dynamic Set Sampling (DSS).

→ DSS: The cache behaviour can be approximated with only a few sets.

→ This way we dont have to maintain a full ATD, instead just maintain the ATD for the subsets.

2) Partitioning algorithm:

$V_{tot} = UA_1^i + UB_1^{16-i} \quad \forall i \in (1, 16-1)$

→ UA_1^i will give the utility info (#misses) based on the UMON-DSS when the partition is done at i^{th} way.

→ The i with the maximum V_{tot} will be chosen as the partition.

? After certain interval, the hit counters in all the UMON are halved ? What does this mean ?

③ Replacement engine:

→ 1 extra bit is append to TAGs indicating the core which the block belongs to based on the partitioning algorithm

→ On conflict miss, count the blocks which belong to the missed block's core.

* If its less than allotted, evict a LRU block from the other core blocks
* If its greater, then evict a LRU block from the current core block.

Scalability:

→ The above mentioned partitioning techniques work only when there are only 2 application but gets difficult to scale.

↳ Greedy Algorithm
↳ Look Ahead Algorithm.

Here's how the utility (benefit) for both applications changes as they get more cache blocks:

Cache Blocks	App A (Miss Reduction)	App B (Miss Reduction)
1	1	2
2	3	3
3	6	5
4	7	6
5	8	8
6	9	9

This table shows that App A and App B experience different improvements in miss reduction as they receive more cache blocks.

Greedy Algorithm Example:

1. **Initial Setup**: Both App A and App B start with 0 cache blocks.
2. **First Allocation**: The algorithm looks at the benefit of giving 1 block to either App A or App B:
   - App A gets a miss reduction of 1.
   - App B gets a miss reduction of 2.
   - The algorithm allocates the first block to App B since it has a higher immediate benefit.
3. **Second Allocation**: Now, App B has 1 block. The algorithm looks again:
   - App A gets a miss reduction of 1 with 1 block.
   - App B gets an additional miss reduction of 1 (total reduction is now 3).
   - The algorithm still favors App B because both have the same benefit, but App B is chosen arbitrarily in the case of a tie.
4. **Third Allocation**: The process repeats, and App B keeps getting blocks as long as it shows equal or higher benefit than App A.

With this greedy approach, the algorithm makes short-term decisions based on immediate gains without looking ahead at how giving blocks to App A might eventually yield a higher total reduction.

Lookahead Algorithm Example:

The lookahead algorithm, on the other hand, takes into account **future utility** for the next few blocks, not just the immediate benefit.

1. **First Allocation**: The lookahead algorithm checks what happens if it gives more blocks to either App A or App B:
   - If App A gets the next block, its total reduction will increase to 3 if it receives 2 blocks.
   - If App B gets the next block, its reduction will increase to 5 if it receives 3 blocks.
   - The lookahead algorithm evaluates the cumulative benefit (over several blocks) and sees that while App B still has better immediate benefits, App A will catch up after receiving 3 blocks. So, it might give one or two blocks to App A to balance the longer-term gains.
2. **Second Allocation**: After taking future gains into account, it might allocate more blocks to App A earlier than the greedy algorithm would, allowing App A to gain more total cache miss reduction over time.

Mind map:

High demand ≠ Performance improve with bigger cache (Scan)

→ Types of apps
- → Low util
- → High util
- → Saturating util

$U_a^{lb} = miss_a - miss_{lb}$

→ UMON

→ Partitioning algo

→ Replacement engine

UMON → ATD
- → UMON-Local → 1 counter per block
- → UMON-global → 1 counter per way
- → UMON-DSS → UMON-global + DSS

Partitioning algorithm → $U_{tot} = \max_i \left(U_a^i + U_{lb}^{(lb-i)} \right) \quad \forall\, i \in (1, lb-1)$

Replacement engine → Conflict miss → Missing block belongs to app A
- → More block for A than quota → Evict LRU of A
- → Less block for A than quota → Evict LRU of B

Handling scalability (apps > 2)
- → Greedy → At any iter decide whether giving to A or B, based on the overall miss reduction for A or B → Issue → B might completely be ignored even though at some point B might have more hits than A
- → LookAhead → Instead of just deciding now, look ahead few steps and see which gives better