Exploring Cache Hierarchy for Graphs

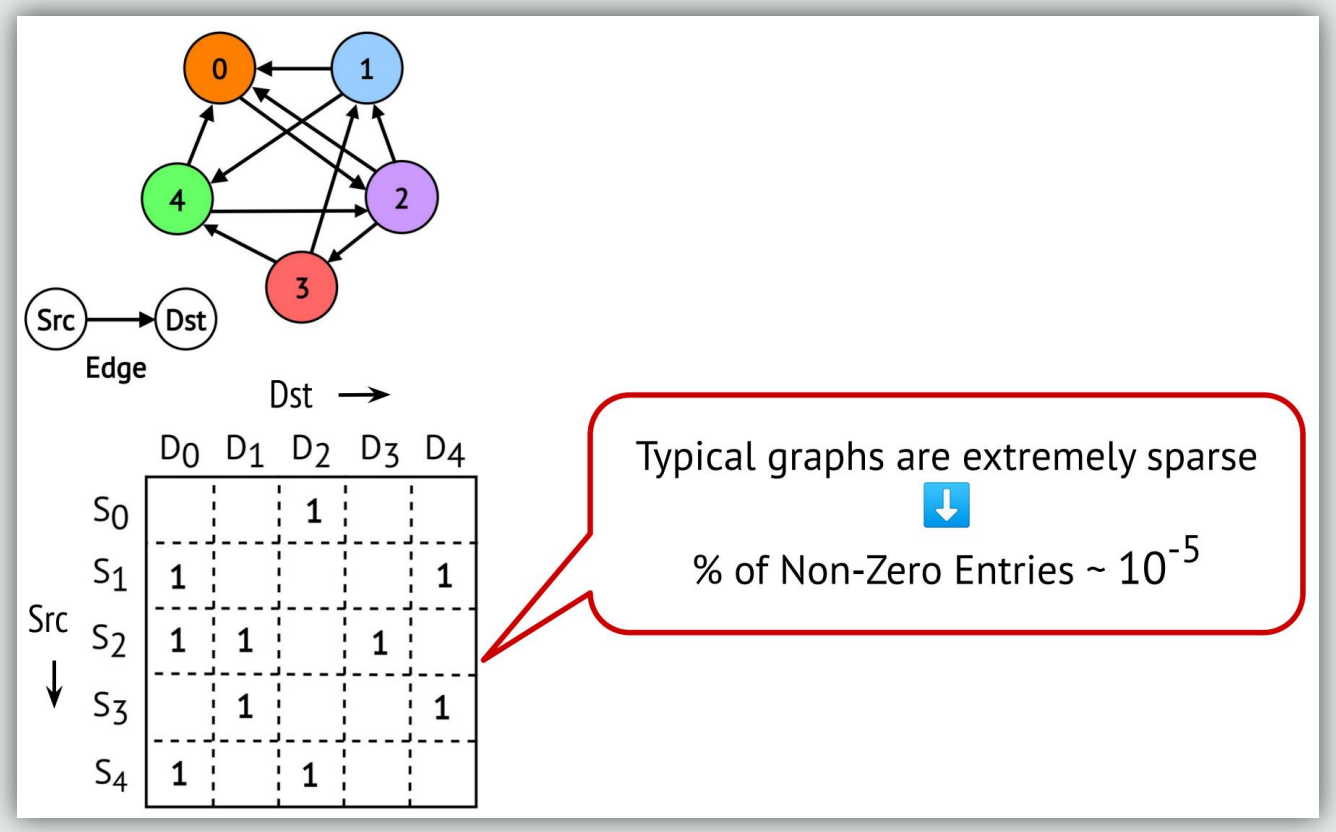
Avadhoot Jadhav

Megh Gohil

Hrishikesh Jedhe Deshmukh

Memory Access Pattern in Graphs

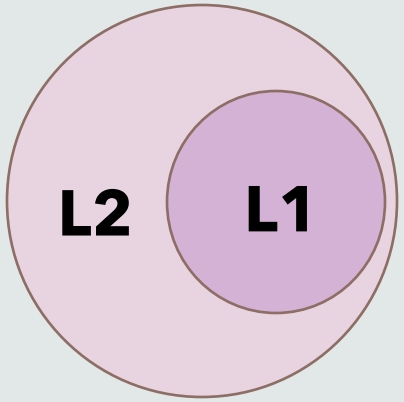
(there isn't any :)



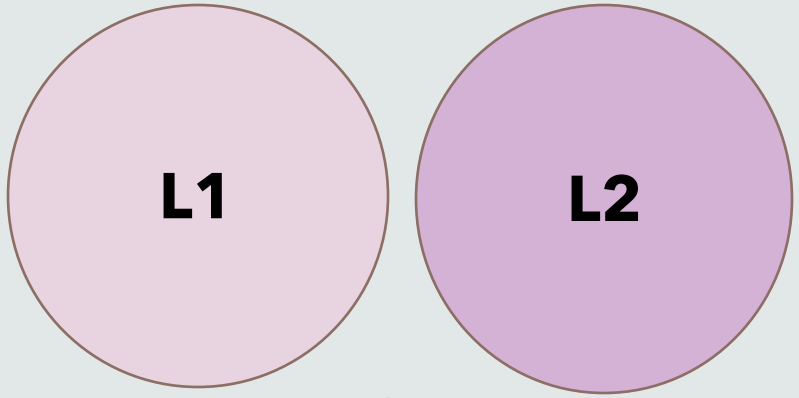
- Sparse storage ⟹ Sparse memory accesses
- Irregular memory accesses leads to poor spatial locality
- Irregular data footprint >>> LLC size
- Not even LLC can contain required memory addresses
- Too many DRAM accesses

Hierarchies

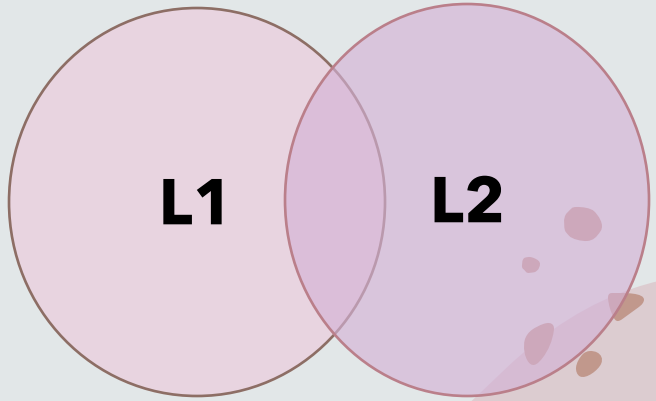
- **Inclusive :** Lower-level caches are super set of upper-level caches
- **Exclusive :** Only 1 copy of memory address in all caches
- **Non-Inclusive Non-Exclusive (NINE) :** Every cache works independently of other caches



Inclusive



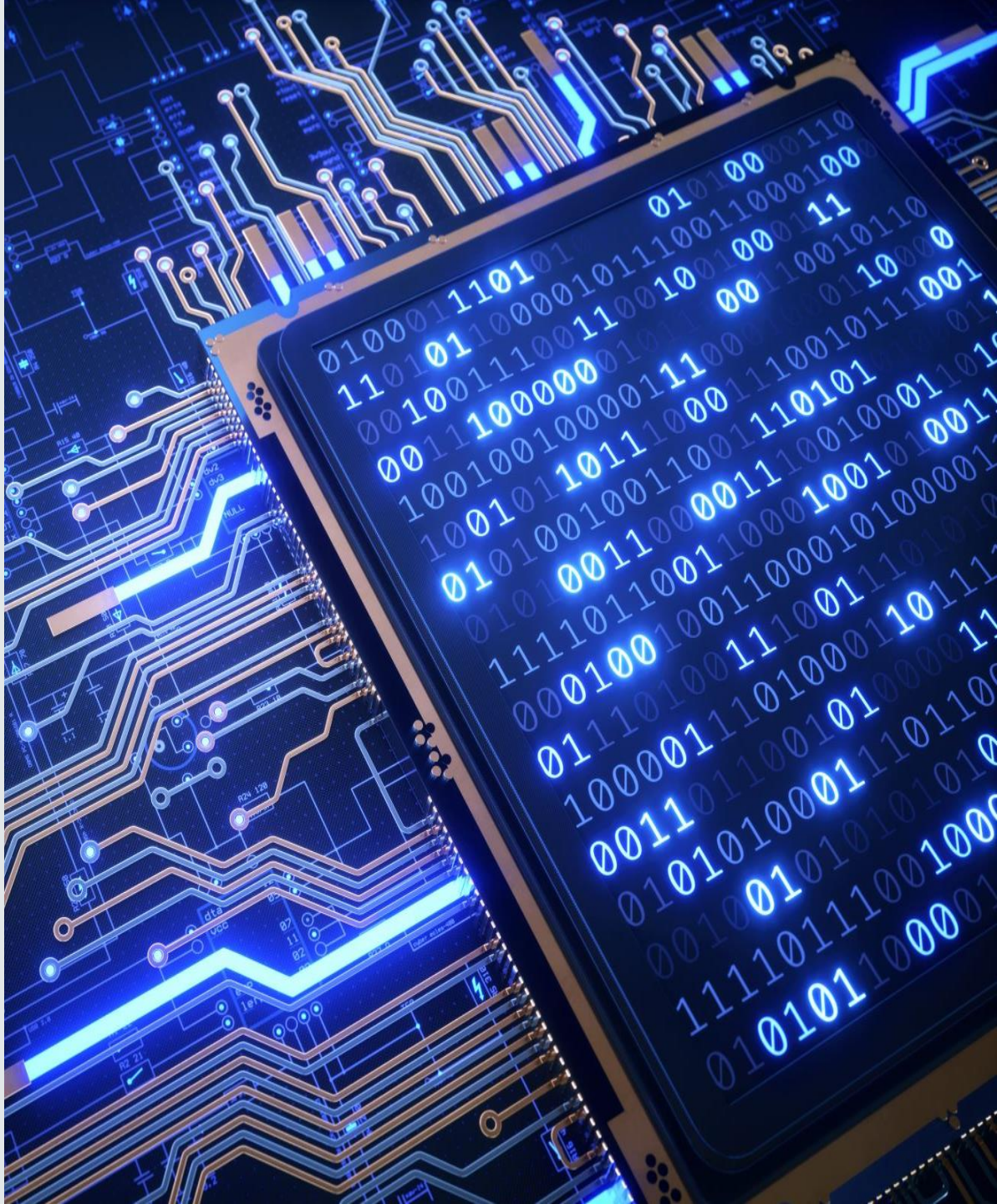
Exclusive



NINE

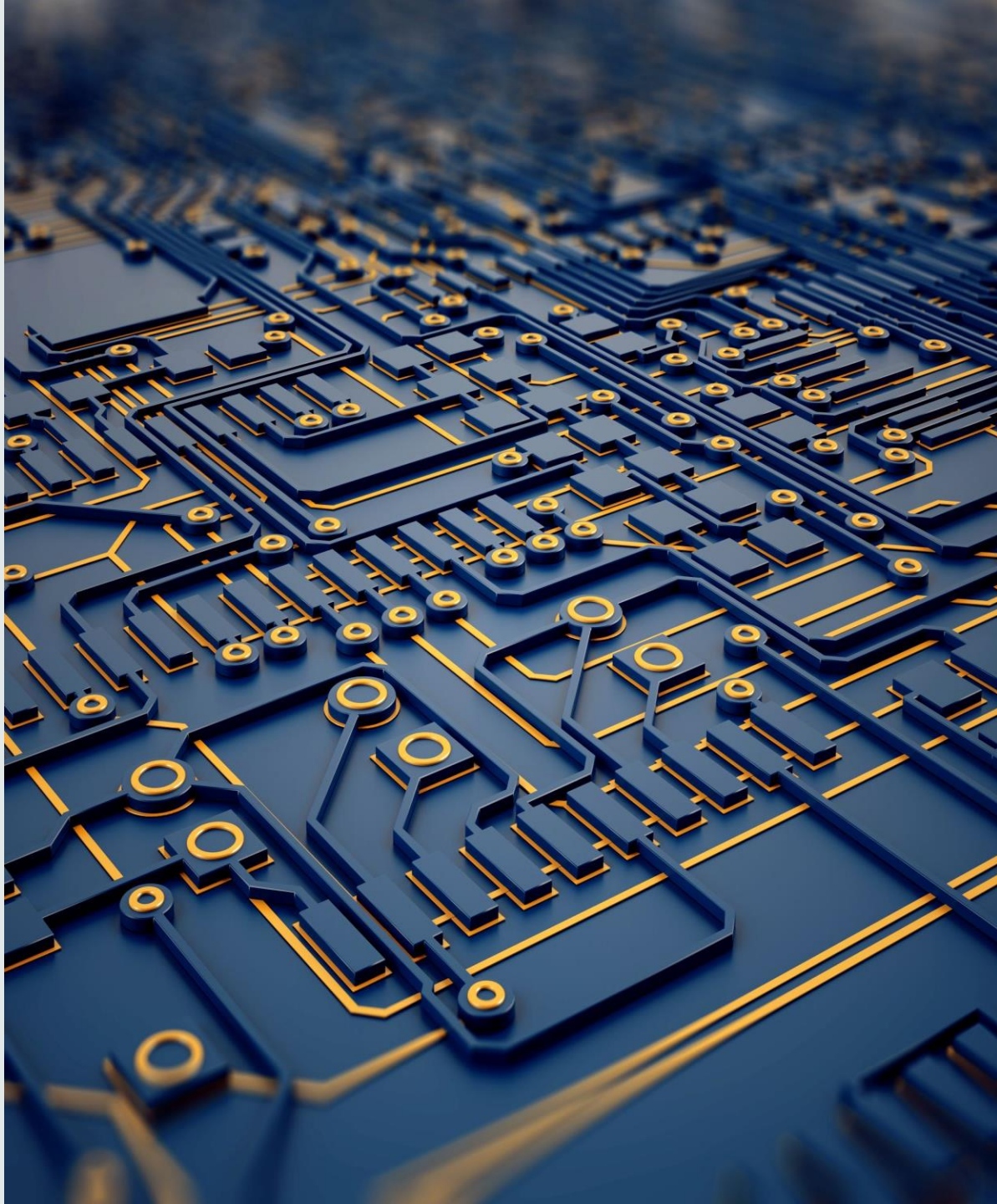
Inclusive Hierarchy

- When certain data is loaded, it is stored in all the caches
- Whenever certain data is evicted from cache, that same data is also evicted from all the higher-level caches
- Extra care needs to be taken while evicting the data from higher level caches when data is dirty



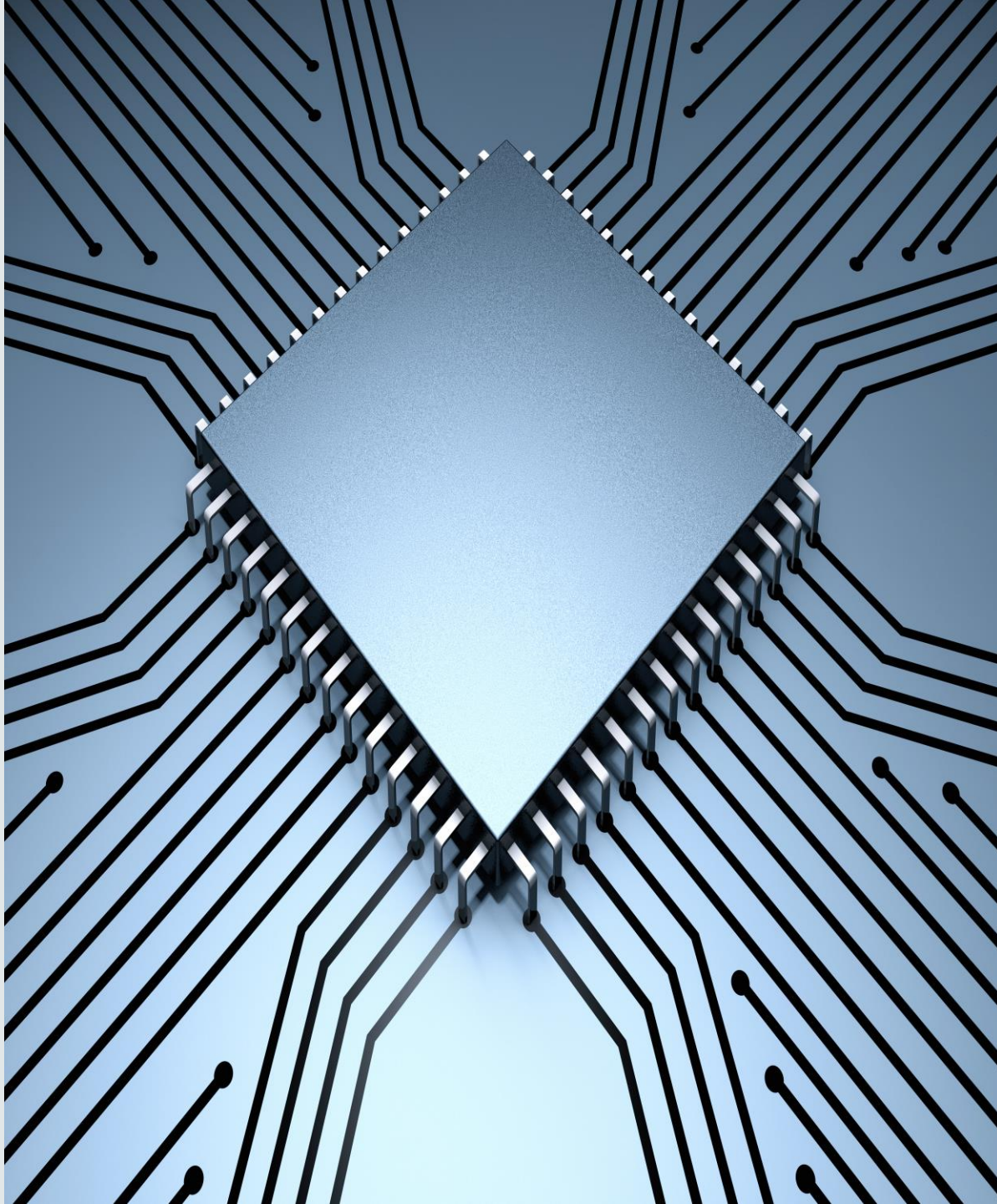
Exclusive Hierarchy

- When data is loaded, it is stored only in the highest-level cache
- When some data is evicted from cache, it is stored into the lower level cache
- Consequence of this is that only 1 copy of each data resides in whole cache hierarchy



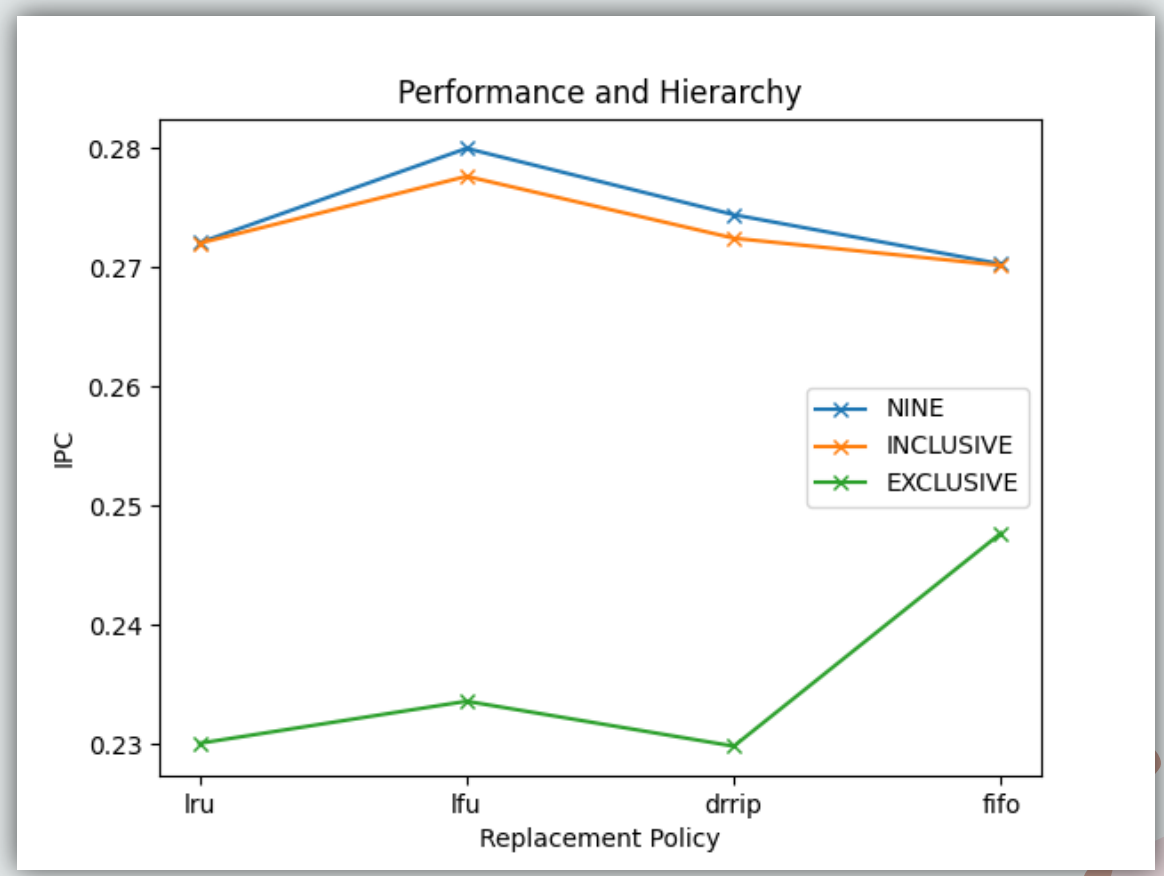
Non-Inclusive
Non-Exclusive

- When data is loaded, it is stored in all the caches
- While evicting data from a cache, no special care needs to be taken
- Simplest policy to implement



Performance Comparison

- Exclusive hierarchy has the worst performance of all
- Inclusive and NINE have almost comparable performance



Why is Exclusive the worst?

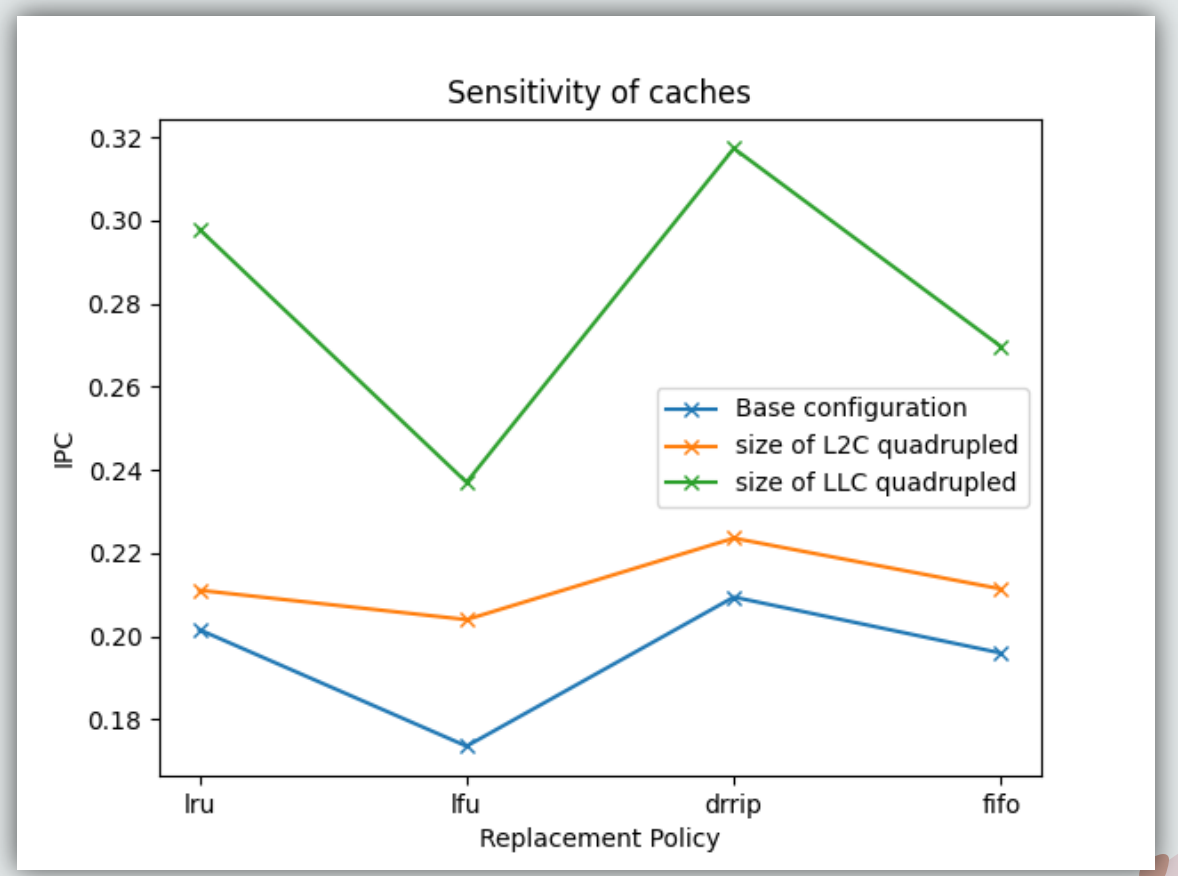
- Due to specification of Exclusive hierarchy, each data address is first loaded into L1 cache (highest-level cache)
- Graph workload having irregular accesses leads to eviction of memory address almost immediately
- Evicted data is again stored into L2, which in turn is immediately evicted and then stored in LLC
- This Load-Evict chain eats up memory bandwidth, thus reducing performance

Why are Inclusive and NINE comparable?

- Only point of difference between INCLUSIVE and NINE is the back-invalidation involved in Inclusive hierarchy
- When some address is evicted from cache, the probability of that address being present in higher level cache is very low due to small sizes of higher-level caches
- So, we rarely need to perform back-invalidation leading to almost same performance of NINE and INCLUSIVE
- NINE outperforms INCLUSIVE by a fraction just because we cannot ignore back-invalidation completely

Sensitivity of Caches

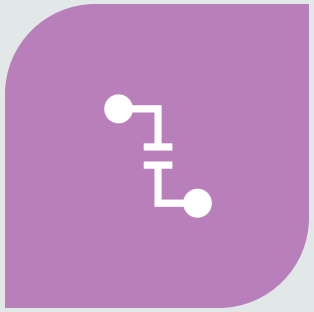
- Once we quadrupled the size of L2C and then quadrupled the size of LLC
- Performance gain from LLC capacity increase is way more than from L2C



LLC more sensitive than L2C. Why?

AS WE SAW AT THE START, NOT EVEN LLC CAN CONTAIN ALL THE MEMORY FOR GRAPH WORKLOAD

SO, L2C BEING EXTREMELY SMALL COMPARED TO LLC, IS COMPLETELY THRASHED AND MOST OF THE ACCESSES GO TO LLC FOR MEMORY REQUESTS

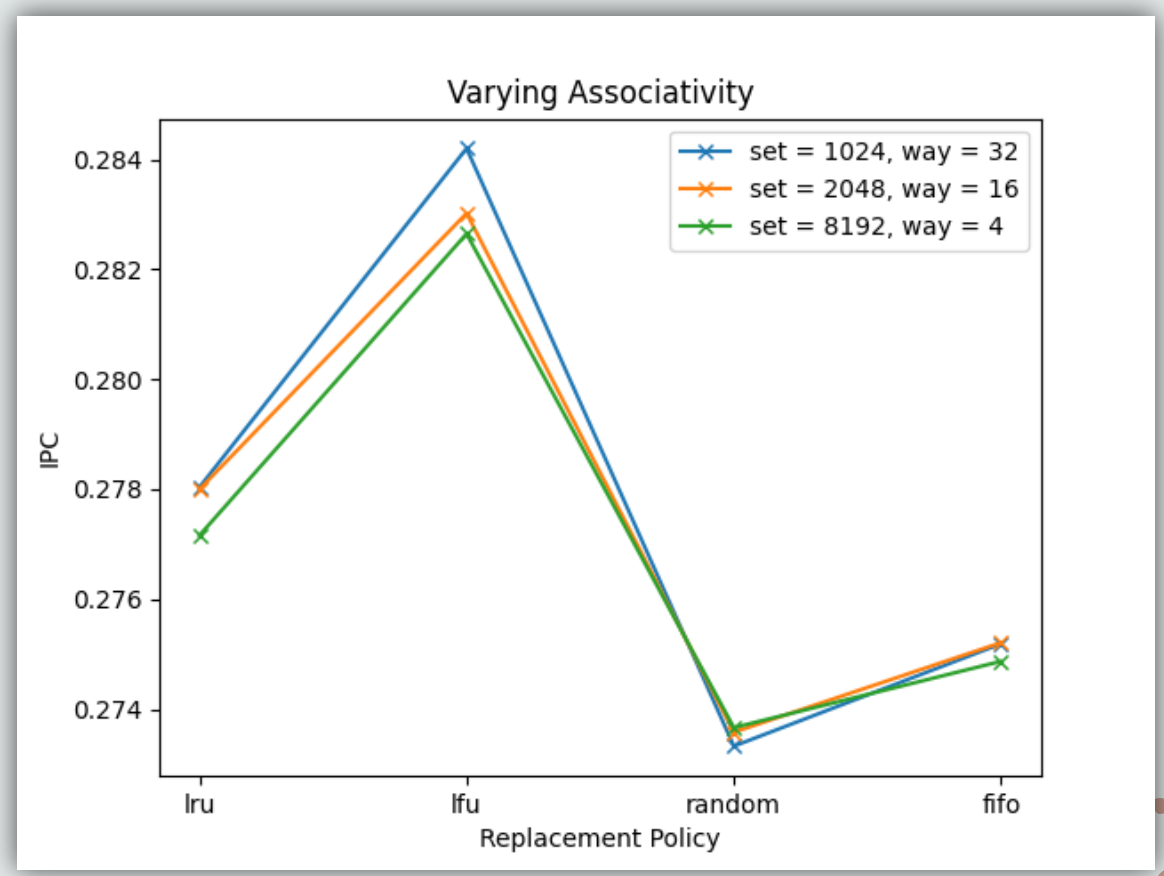


AS MOST OF MEMORY ACCESS REQUESTS ARE HANDLED BY LLC, ITS SIZE IS A BIGGER FACTOR IN DETERMINING PERFORMANCE THAN THE SIZE OF L2C

WE CAN SAY LLC IS MORE "SENSITIVE" THAN L2C FOR GRAPH WORKLOAD

Effect of Associativity

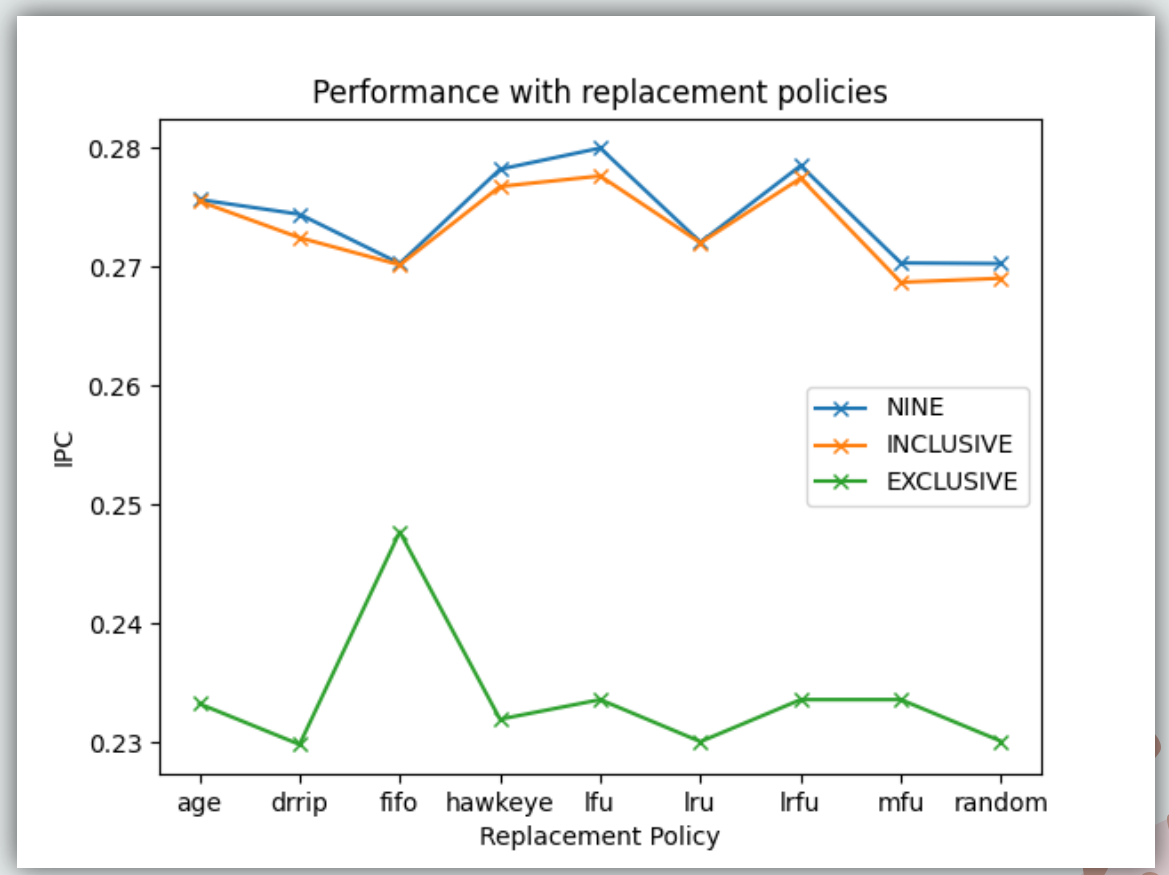
- Varied associativity keeping original size same
- Negligible effect on performance
- Reason for comparable performance is that the whole LLC is thrashed so not even greater associativity can avoid conflict misses




Onto Replacement Policies

Relative Performance of Replacement Policies

- Age, Hawkeye, LRFU and MFU are the newly implemented replacement policies
- As expected, EXCLUSIVE is the worst for all policies



Hawkeye

- Based on Belady's Algorithm (sadly can't predict the future)
- Uses past memory accesses to predict what future accesses may look like
- Once we predict the future accesses, we approximately use Belady's Algorithm to determine victim to evict
- It is still giving less performance than good old LFU :(
- Possible reason is that the graph workload is so irregular that past accesses are not a good indicator of future accesses

LRU + LFU (LRFU)

(oh we ran out of names)

We thought of combining both LRU + LFU policies

LRFU takes into consideration both Recency and Frequency of usage

Let the score associated with a line be ***a.(NUM_WAY - lru) + b.(frequency)***, where "a" and "b" act as weights to determine importance of recency and frequency

The cache line with the least score is evicted

This policy has better performance than LRU and has almost comparable performance with LFU :)

LFU with Ageing Counter

- Next, we implemented a modified version of LFU. We associated an ageing counter with each cache line
- If cache line age exceeds a certain threshold, we will evict that line instead of the least frequently used line
- Deciding this threshold is a big challenge and it varies with traces
- Overall, this policy performs a bit worse than LFU, partly due to us not being able to decide on an optimum age threshold

Most Frequently Used (MFU)

(oh so weird)

- We implemented this counter-intuitive policy to highlight an important fact about graph workloads
- One might think that MFU will perform way worse than other policies
- But surprisingly, it only performs fractionally worse than other popular policies
- This observation highlights that in graph workloads, frequency might not be a good indicator of future accesses after all !!!