

Doppelgänger : Cache that hoodwinks

By Akshay Gadre

Aim

- This cache is aimed to reduce the area and energy usage of LLCs by introducing tolerance in the data made available.
- Thus, it is a value based optimization versus reference based optimization.
- Aims to utilize similarity amongst cache data blocks to reduce the data needed to be stored.
- Thus, there exists a tolerance value to be expected.

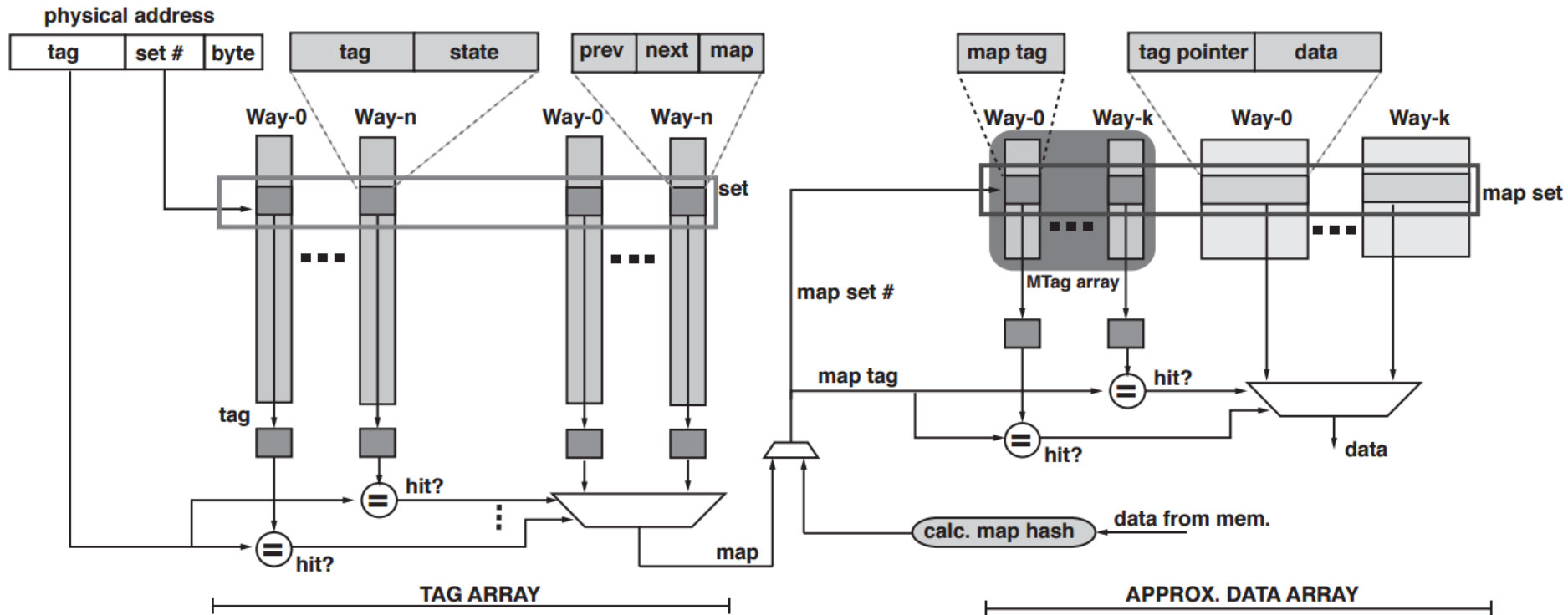
Architecture

- Tag array and Data array exist independently with a **many-one mapping** from tag to map bits
- Multiple tag entries point to the same map which is mapped to a approximate data entry.
- Tag entry contains:
 - Address tag
 - Line's state
 - Prev and Next tag pointers
 - Map value

Architecture(contd.)

- Approximate data entry contains:
 - Map tag
 - Tag Pointer
 - Data Block
- The approximate data entry is indexed with map value instead of the physical address value

Architecture(contd.)



Read

- Input: Address A
- Compare the tag,
 - If not found -> MISS
 - If found ->
 - Use the map field of the tag to index into the MTag array parallelly (guaranteed to match)
 - Use the data line of that MTag to get the approximate data
- If $4 \times \text{data array entries} < \text{tag array entries}$ then combined lookup cost less than conventional cache

Insertion

- Input: Block B with physical address A
- Use the value to calculate the map M
- If a data block with M exists
 - Allocate tag entry using A and tag map entry as M
 - Add tag entry of A to that data block pointed to by M
 - The tag pointer in the data block now points to this tag entry

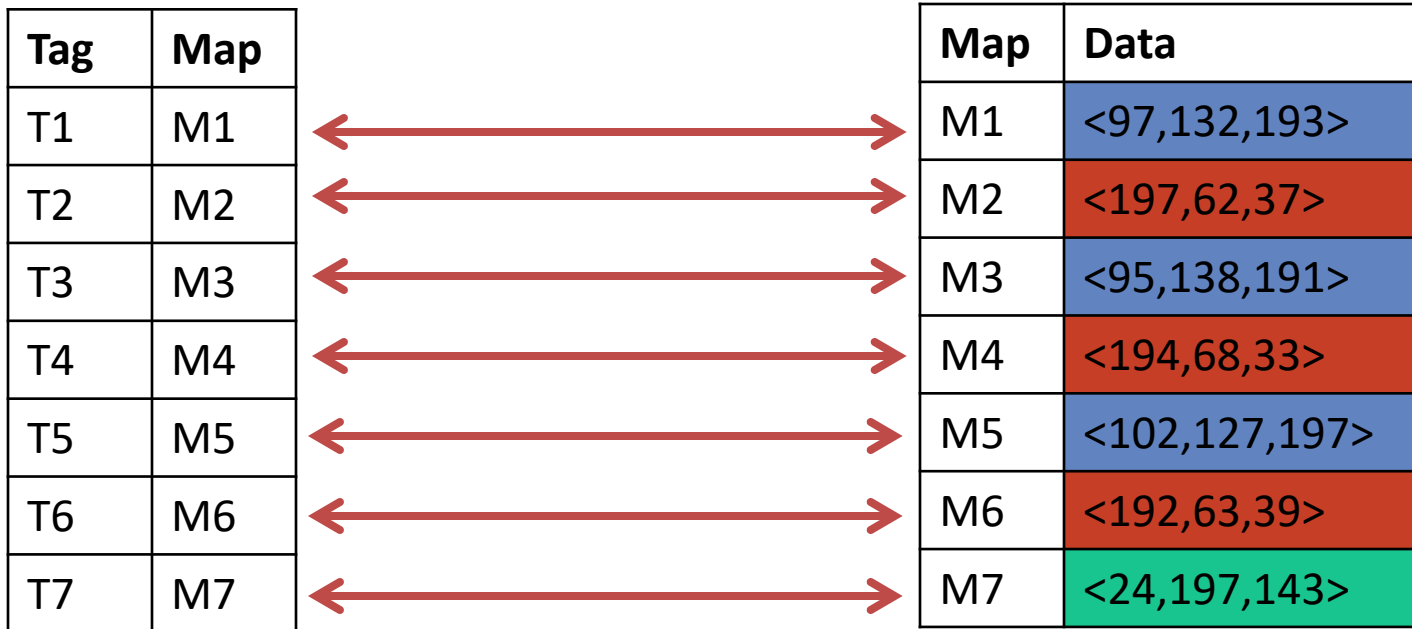
Insertion(contd.)

- If a data block with M does not exist
 - Pick a data block D to evict
 - Invalidate all tag entries which point to D and write-back any dirty data (Also take care of inclusiveness if required)
 - Add the current data block with the new tag and make the prev and next tag as NULL.
 - Map tag field is set to M

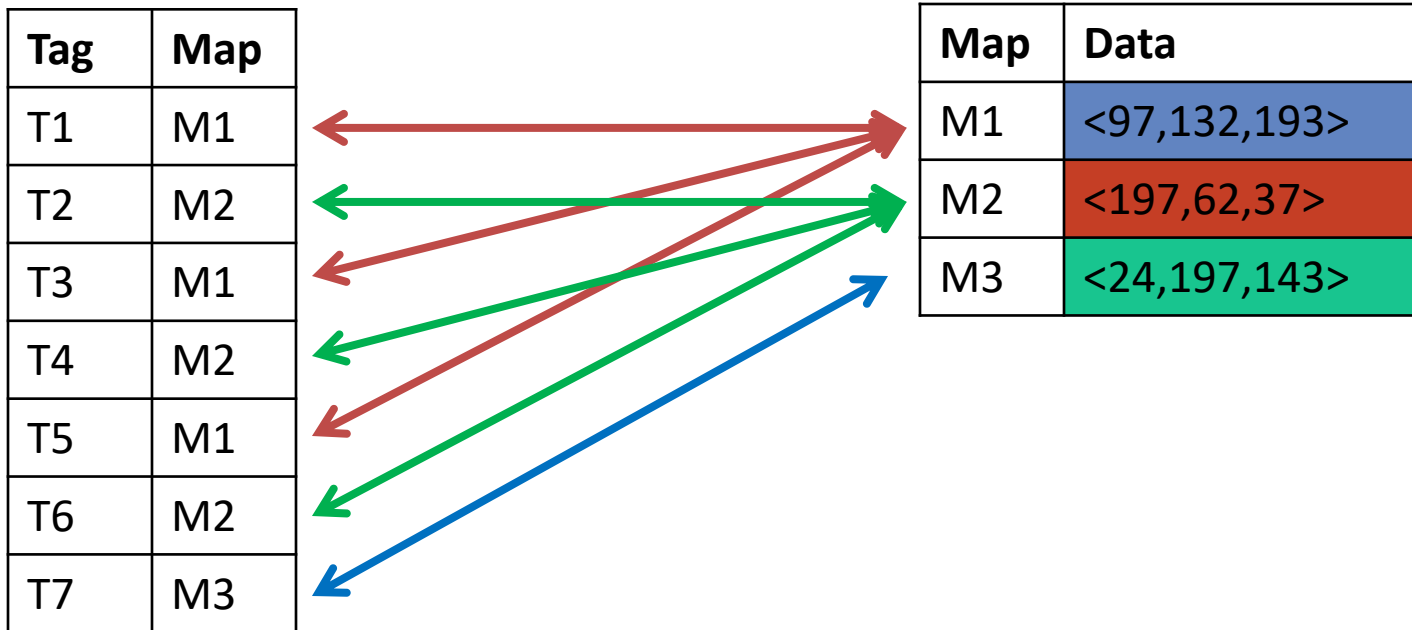
Write

- Input: Some write to a physical address A
- Recompute the map using the updated data for the block
 - If $\text{map}_{\text{new}} = \text{map}_{\text{old}}$ then set dirty bit = 1
 - If $\text{map}_{\text{new}} \neq \text{map}_{\text{old}}$ then
 - See if map_{new} entry already exists
 - If yes, Remove the current data from the map_{old} linked list and add to map_{new} linked list
 - If no, perform insertion for this block with map_{new}
 - In both cases, set the dirty bit = 1

Conventional LLC Mapping

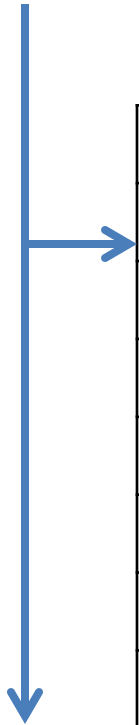


Doppelgänger LLC Mapping



Approximation cache such as Doppelgänger trades off accuracy for saving up space and reducing latency of the LLC and works immensely well for specific applications like graphics and video

Doppelgänger LLC Lookup

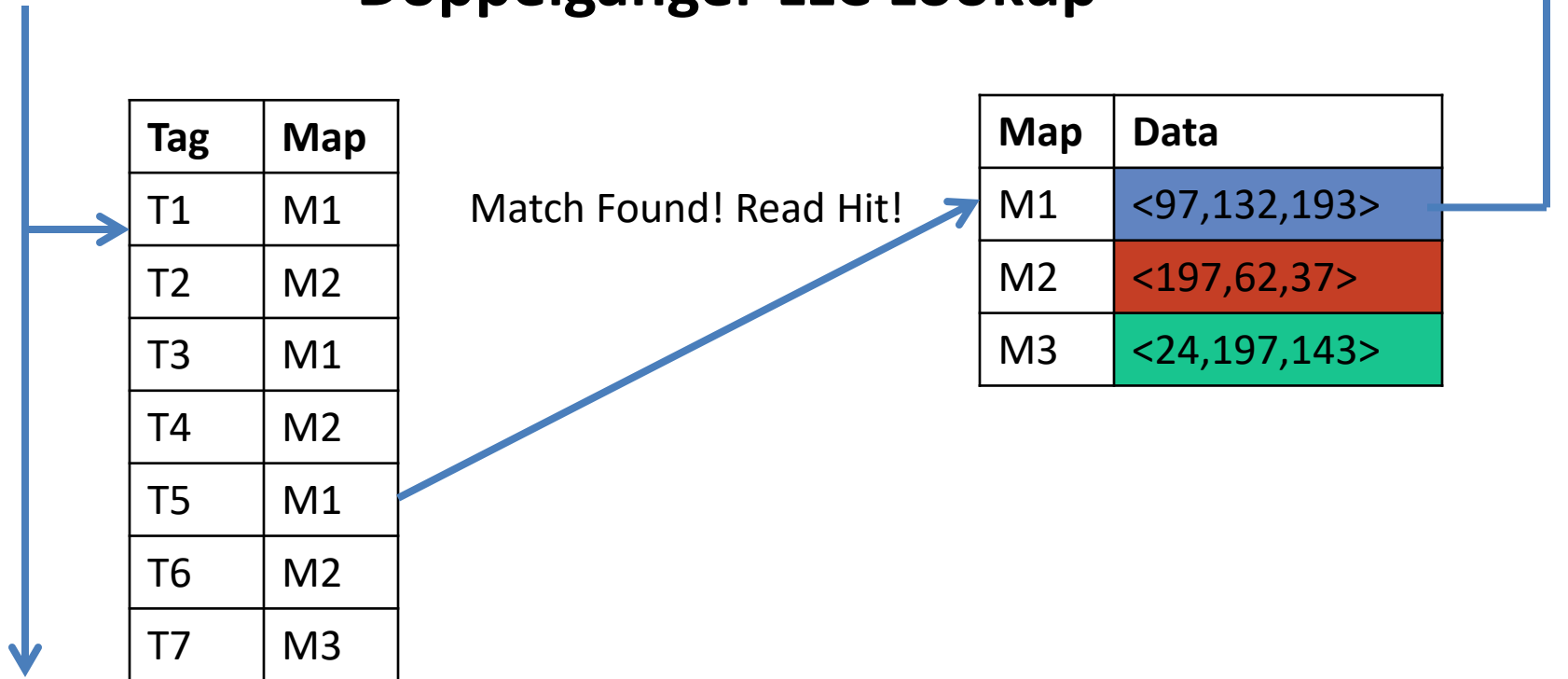


Tag	Map
T1	M1
T2	M2
T3	M1
T4	M2
T5	M1
T6	M2
T7	M3

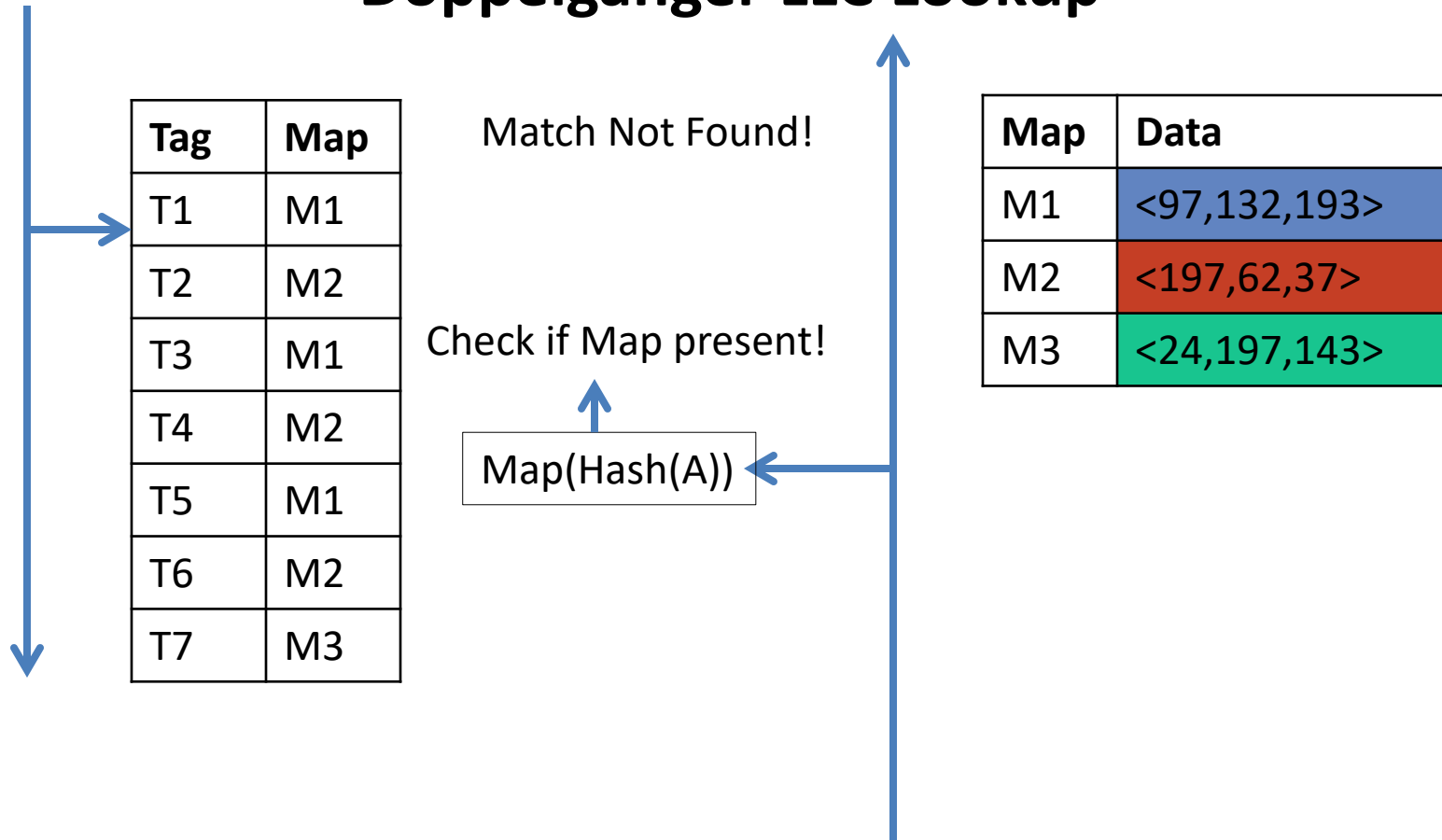
Map	Data
M1	<97,132,193>
M2	<197,62,37>
M3	<24,197,143>

Send Data Back to the user

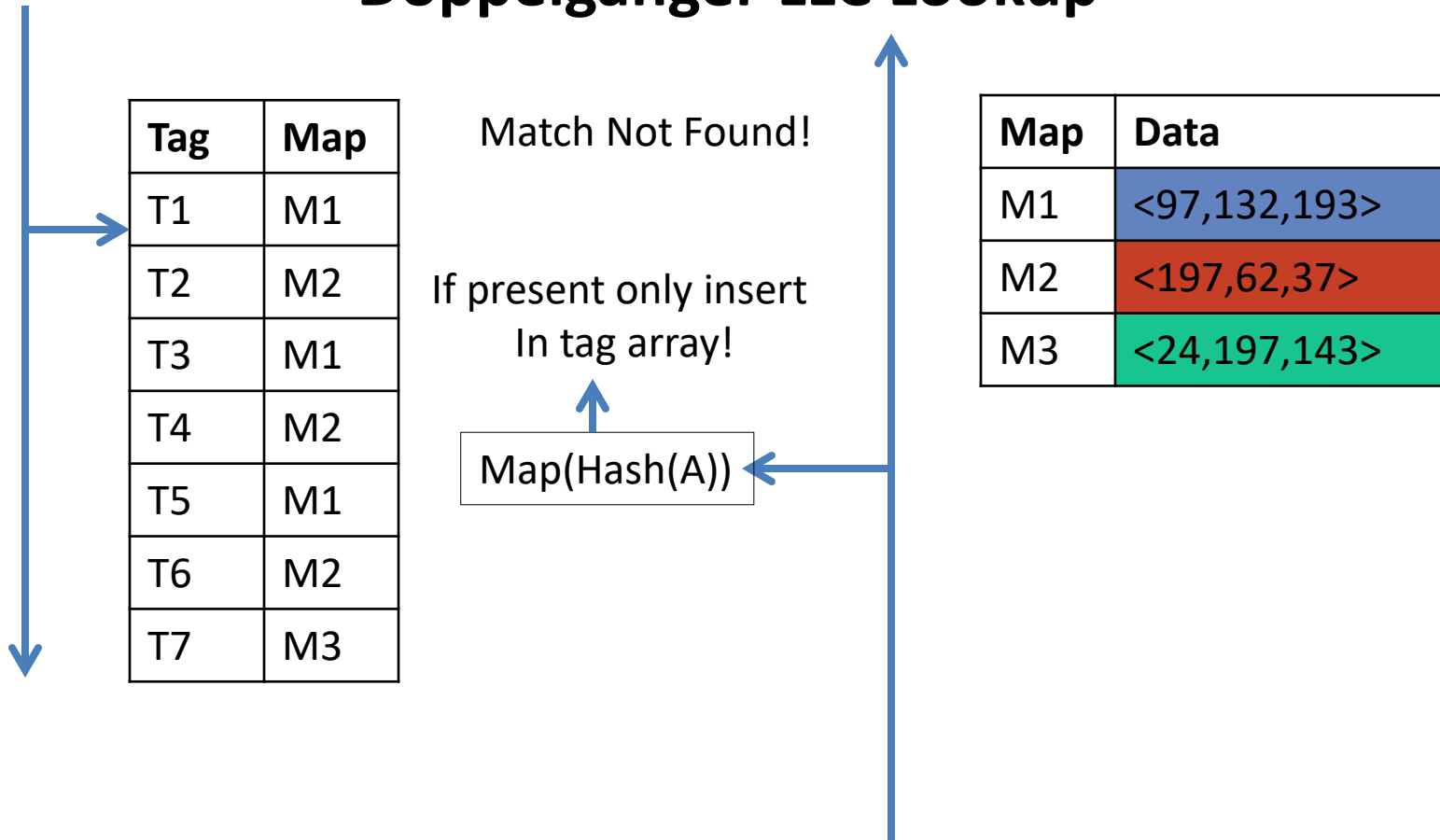
Doppelgänger LLC Lookup



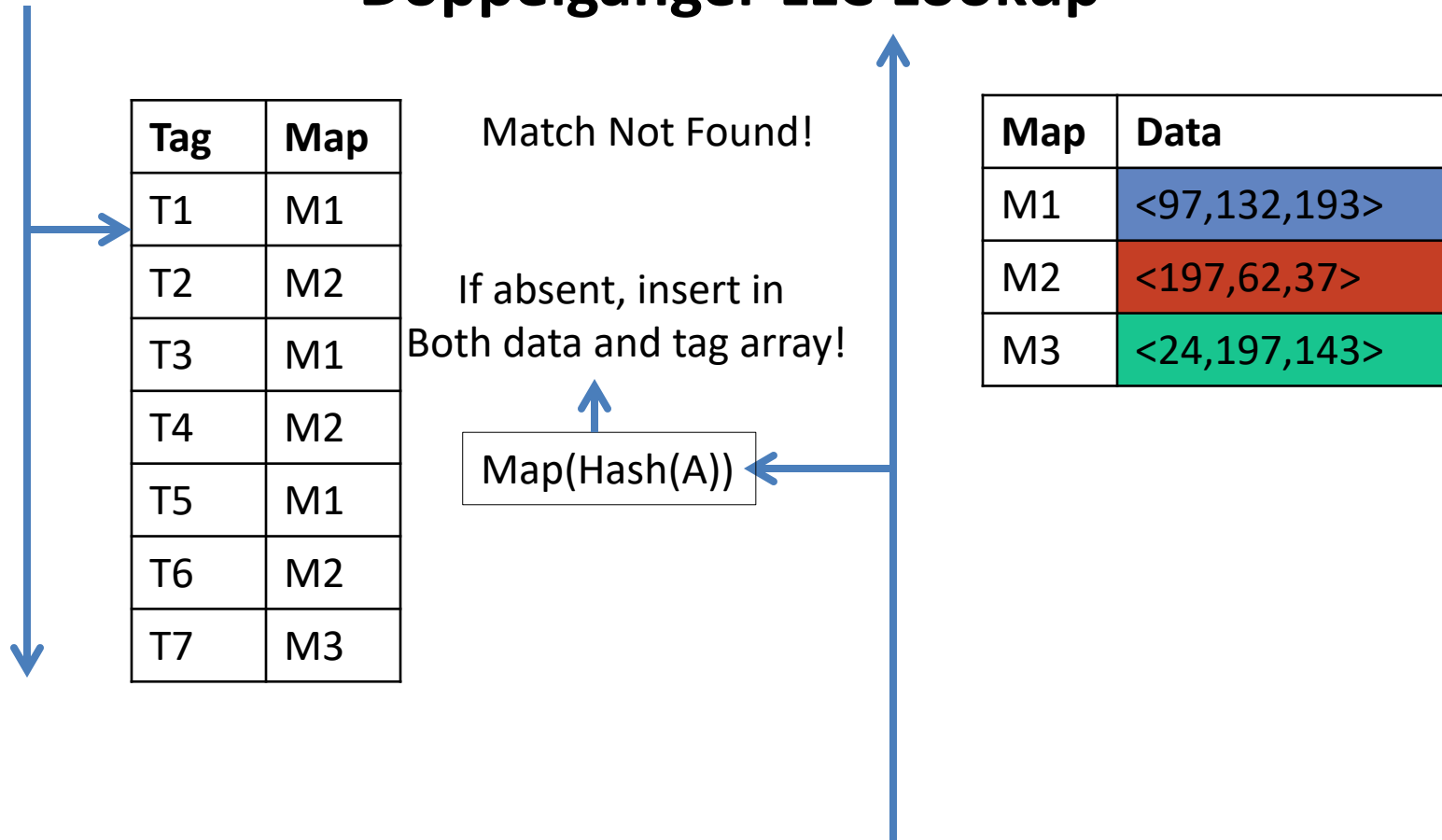
Doppelgänger LLC Lookup



Doppelgänger LLC Lookup



Doppelgänger LLC Lookup



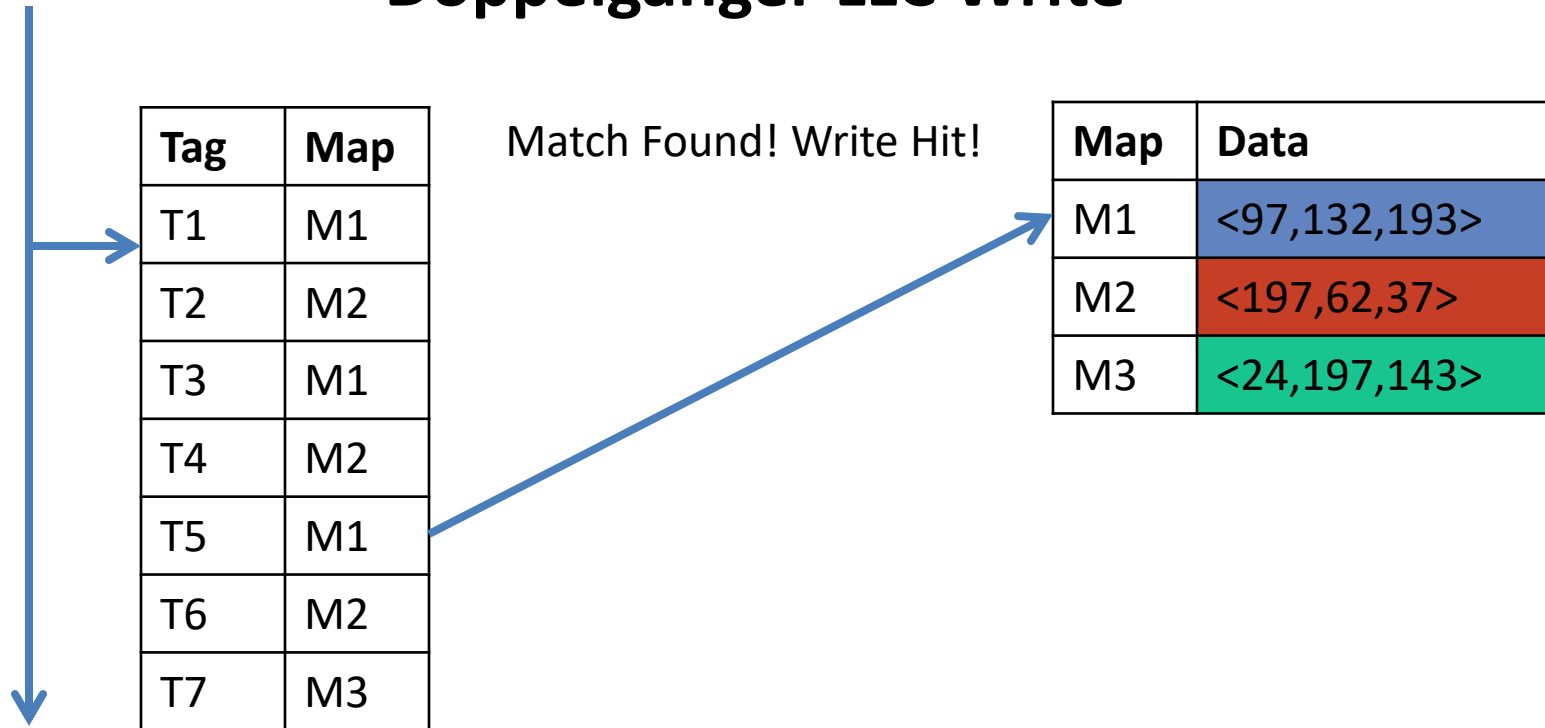
Doppelgänger LLC Write



Tag	Map
T1	M1
T2	M2
T3	M1
T4	M2
T5	M1
T6	M2
T7	M3

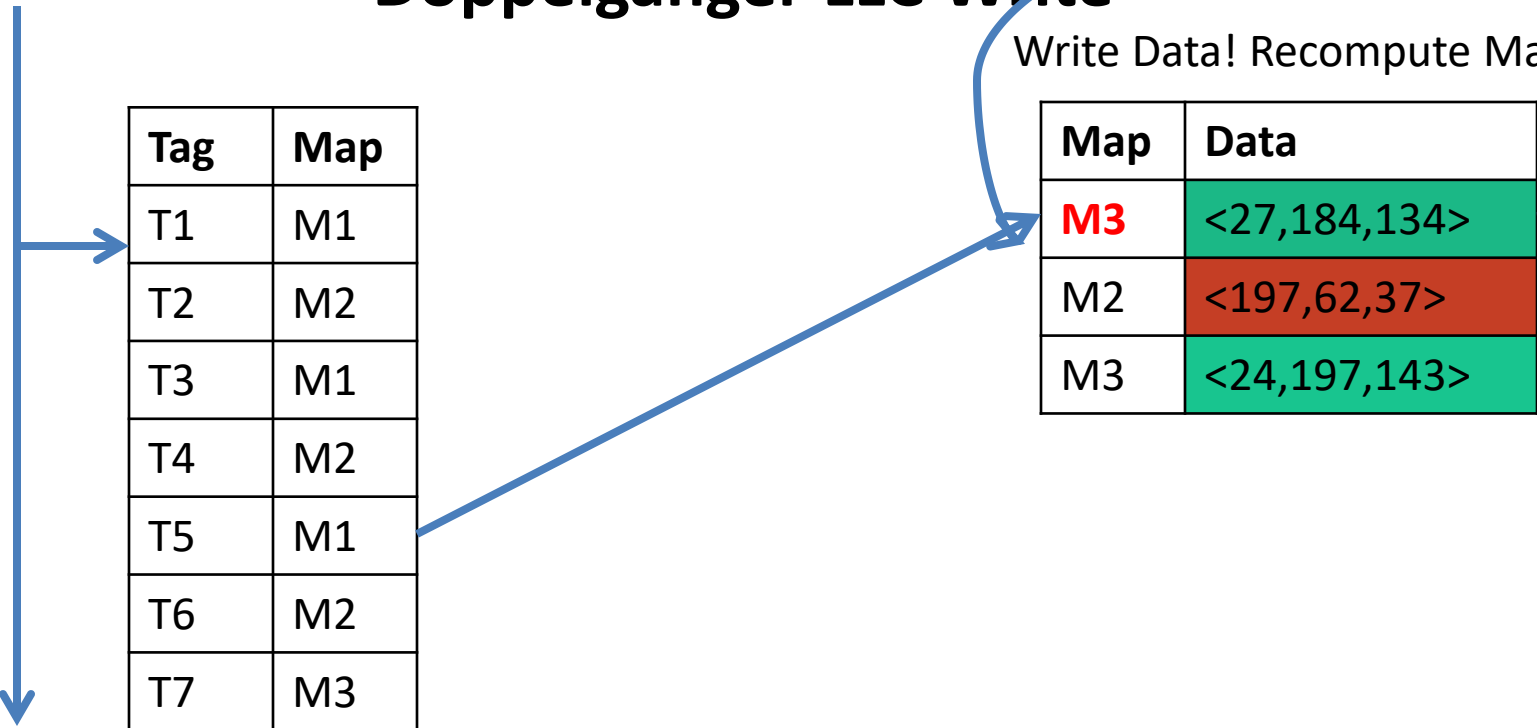
Map	Data
M1	<97,132,193>
M2	<197,62,37>
M3	<24,197,143>

Doppelgänger LLC Write



Doppelgänger LLC Write

Write Data! Recompute Map!

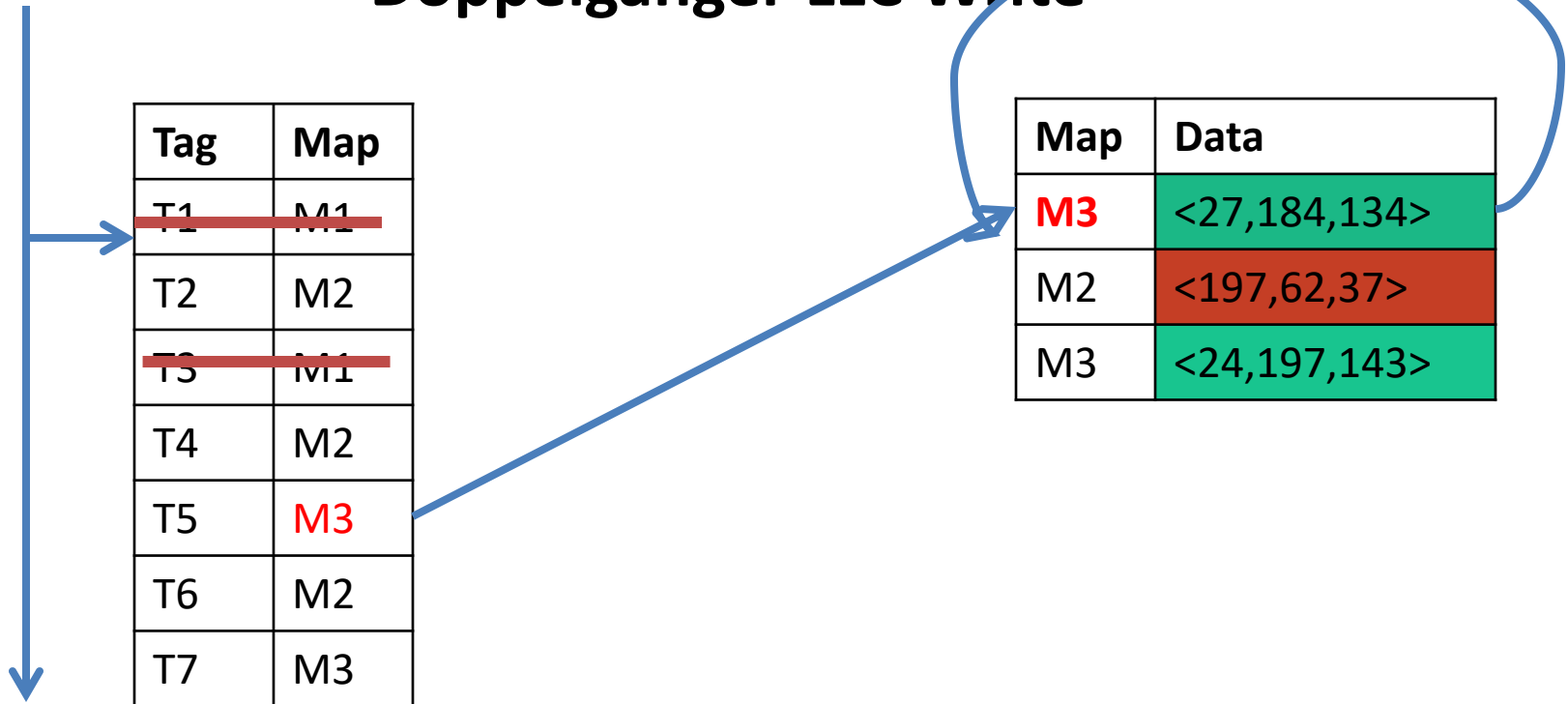


The diagram illustrates a 'Doppelgänger LLC Write' operation. On the left, a table with columns 'Tag' and 'Map' lists seven entries (T1-T7) mapped to three memory sets (M1, M2, M3). A vertical blue arrow on the left points downwards, and a horizontal blue arrow points from the left edge of the table to the right. A blue arrow originates from the 'M1' cell in the T5 row and points to the 'M3' cell in the first row of the table on the right. On the right, a table with columns 'Map' and 'Data' shows the current state of the memory sets. The 'M3' entry in the first row is highlighted in red, and the 'M2' entry in the second row is highlighted in red. The 'M1' and 'M3' entries in the first and third rows are highlighted in green. A blue oval encircles the text 'Write Data! Recompute Map!' and the right table.

Tag	Map
T1	M1
T2	M2
T3	M1
T4	M2
T5	M1
T6	M2
T7	M3

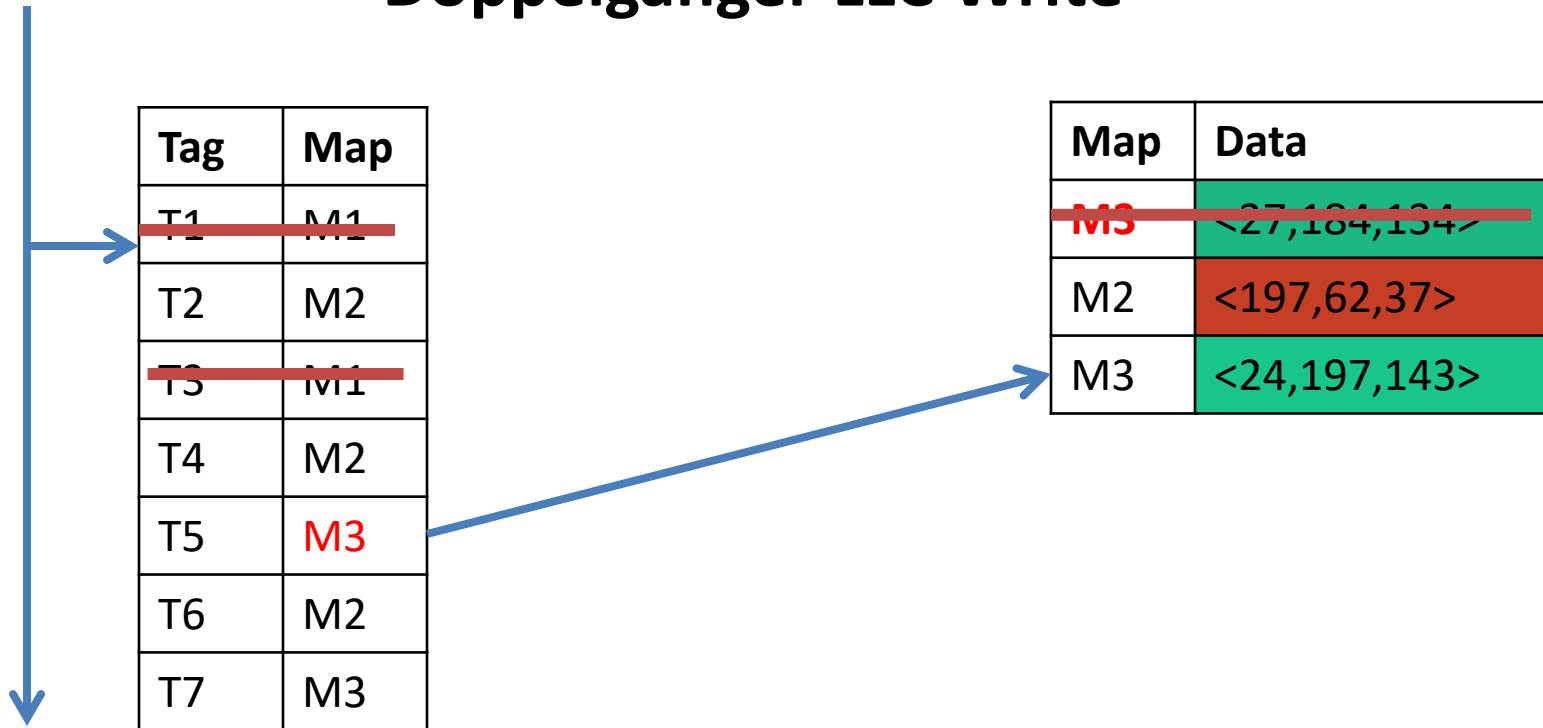
Map	Data
M3	<27,184,134>
M2	<197,62,37>
M3	<24,197,143>

Doppelgänger LLC Write




Invalidate all tags and do their write-backs for write-back caches except the one being written to

Doppelgänger LLC Write




If the map already exist remap and delete the current block

Doppelgänger LLC Write



Tag	Map
T1	M1
T2	M2
T3	M1
T4	M2
T5	M1
T6	M2
T7	M3

Map	Data
M1	<97,132,193>
M2	<197,62,37>
M3	<24,197,143>

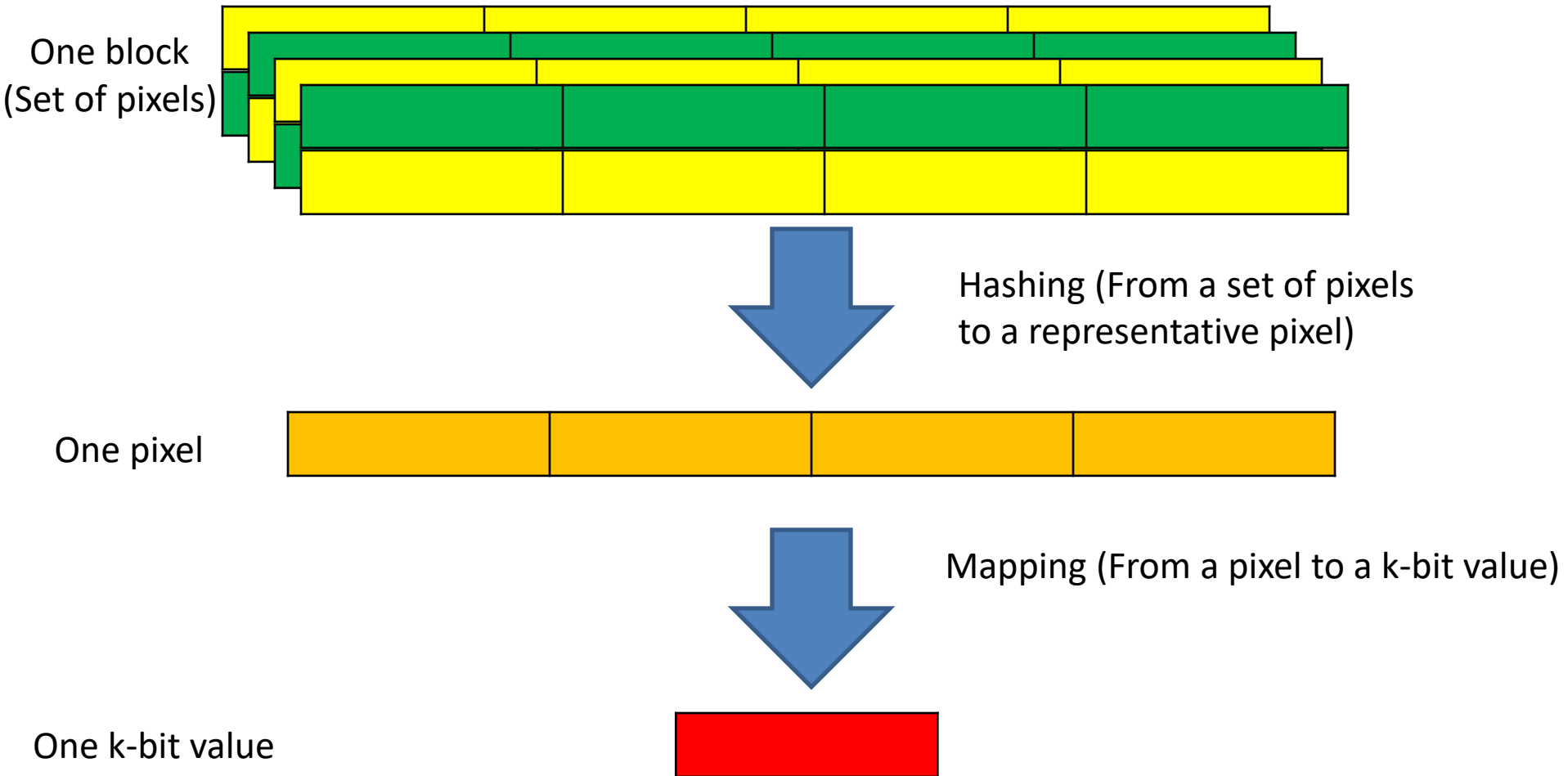


Match Not Found! Do the same algorithm for found only that the data insertion will be done before it

Replacement

- Tag Eviction
 - Tag pointers are nulled (Inclusiveness taken care)
 - If a tag is evicted and only it is pointing to a data block then the data block is also evicted
- Data Eviction
 - All tags associated must be evicted
 - Note that the large no. of recursive invalidations are *off the critical path per se*

Doppelgänger Mapping Algorithm



Doppelgänger Mapping Algorithm

- The mapping is a two-stage process.
- Given a 2^k -byte sized block we get 2^{k-2} pixels per block.
- Each pixel is depicted by the value of its RGB component with a filler block which could otherwise be used for opacity for other use cases.
- First part of the mapping process is the hashing where each of the blocks is depicted as one pixel

Hashing Functions

- Option (a):
 - For each of the channel we take the average of the values of that channel of all pixels in the block
- Option (b):
 - For each of the channel we take the maximum of the values of that channel of all pixels in the block
- Option (c):
 - For each of the channel we take the minimum of the values of that channel of all pixels in the block

Doppelgänger Mapping Algorithm(contd.)

- Given this hashed pixel, we consider further reduction in the tag of the data block. This resultant tag is called the mTag(mapped Tag).
- Error which was already generated is exponentiated by this reduction much more rapidly.
- The best results will intuitively be when we take the top-n bits of all channels in the mapping criterion

Mapping Functions

- Option(a):
 - $n=2$; 75% reduction in the size of tag
- Option(b):
 - $n=4$; 50% reduction in the size of tag
- Option(c):
 - $n=6$; 25% reduction in the size of tag
- Option(d):
 - $n=8$; 0% reduction in the size of tag

Effects due to size of the block

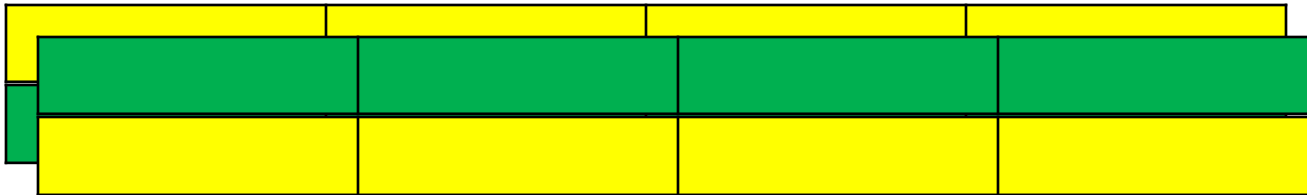
- As is evident from the above explanation, the more the no. of pixels per block, the more is the information distortion at the hashing level.
- Thus the smaller the size of the block the better for hashing accuracy but will also result in higher latency due to increase in the no. of blocks for a given sized cache.
- Thus in approximation caches, the most important currently open problems are of what ratio of tradeoff will be optimum for a given set of settings.

Block Sizes

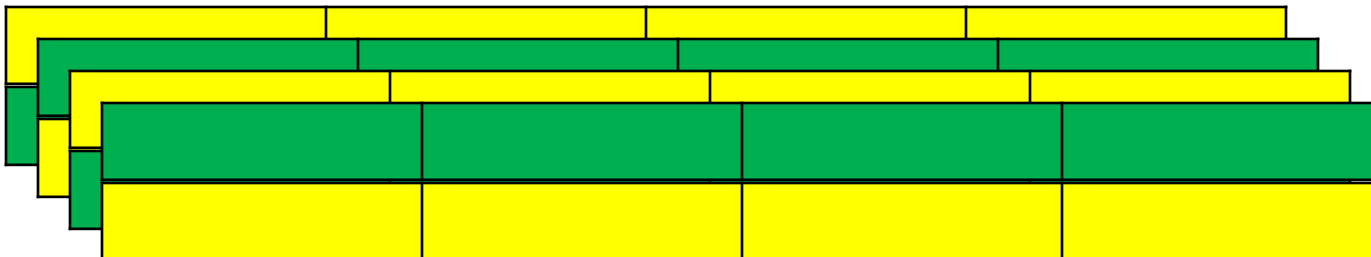
- Option (a): 8 bytes – 2 pixels



- Option (b) 16 bytes – 4 pixels



- Option (a) 32 bytes – 8 pixels



SIMULATIONS AND RESULTS

Simulation Environment

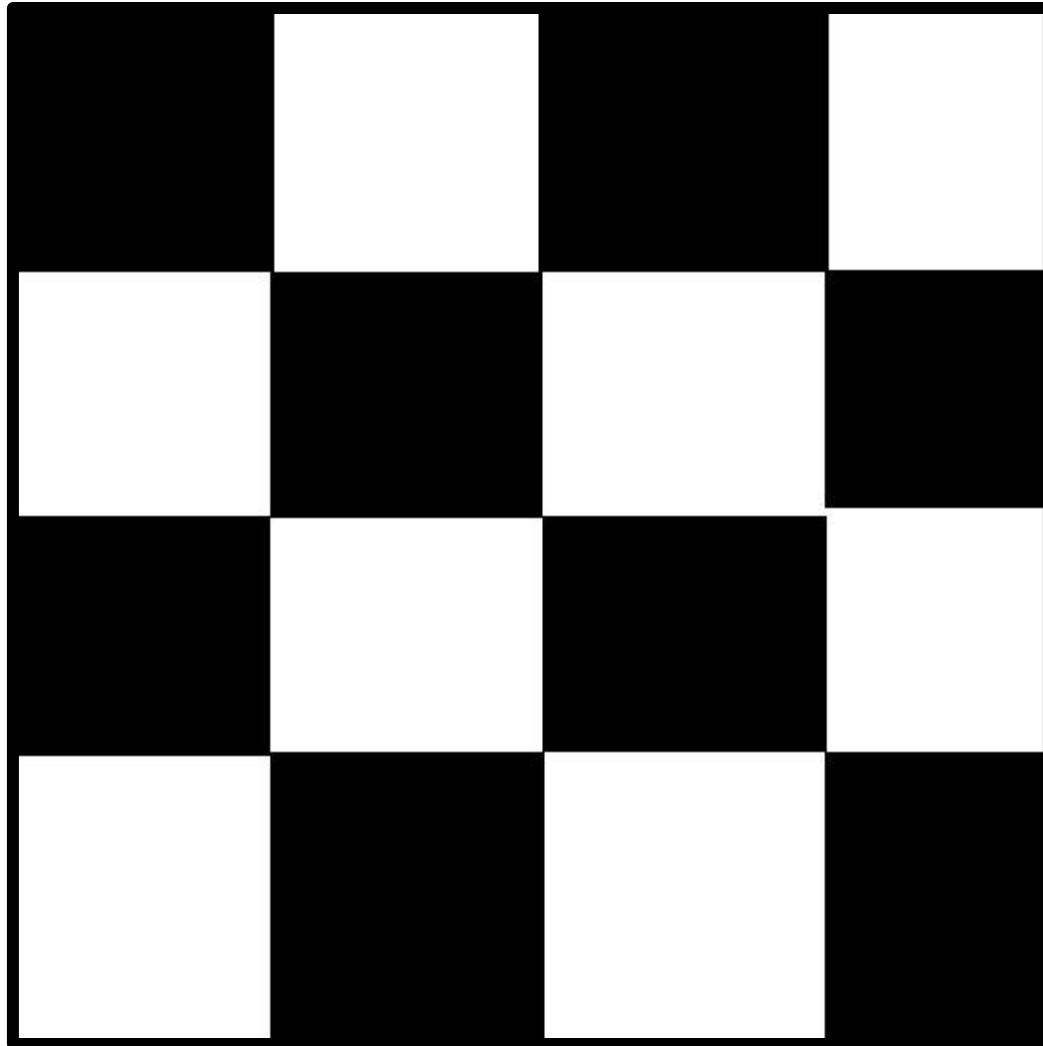
- First to compare this cache with the conventional cache we choose a normal LLC with
 - Associativity=8
 - Size of Block=16
 - Physical Address Space=1048576
 - Cache Size=65536

Simulation Environment(Contd.)

- We have chosen the below 512px * 512px pictures for specific reasons with special qualities in their pixel values.
- Each of these pictures tests a specific parameter which we want to test, i.e., hashFunction, mapFunction, blockSize.

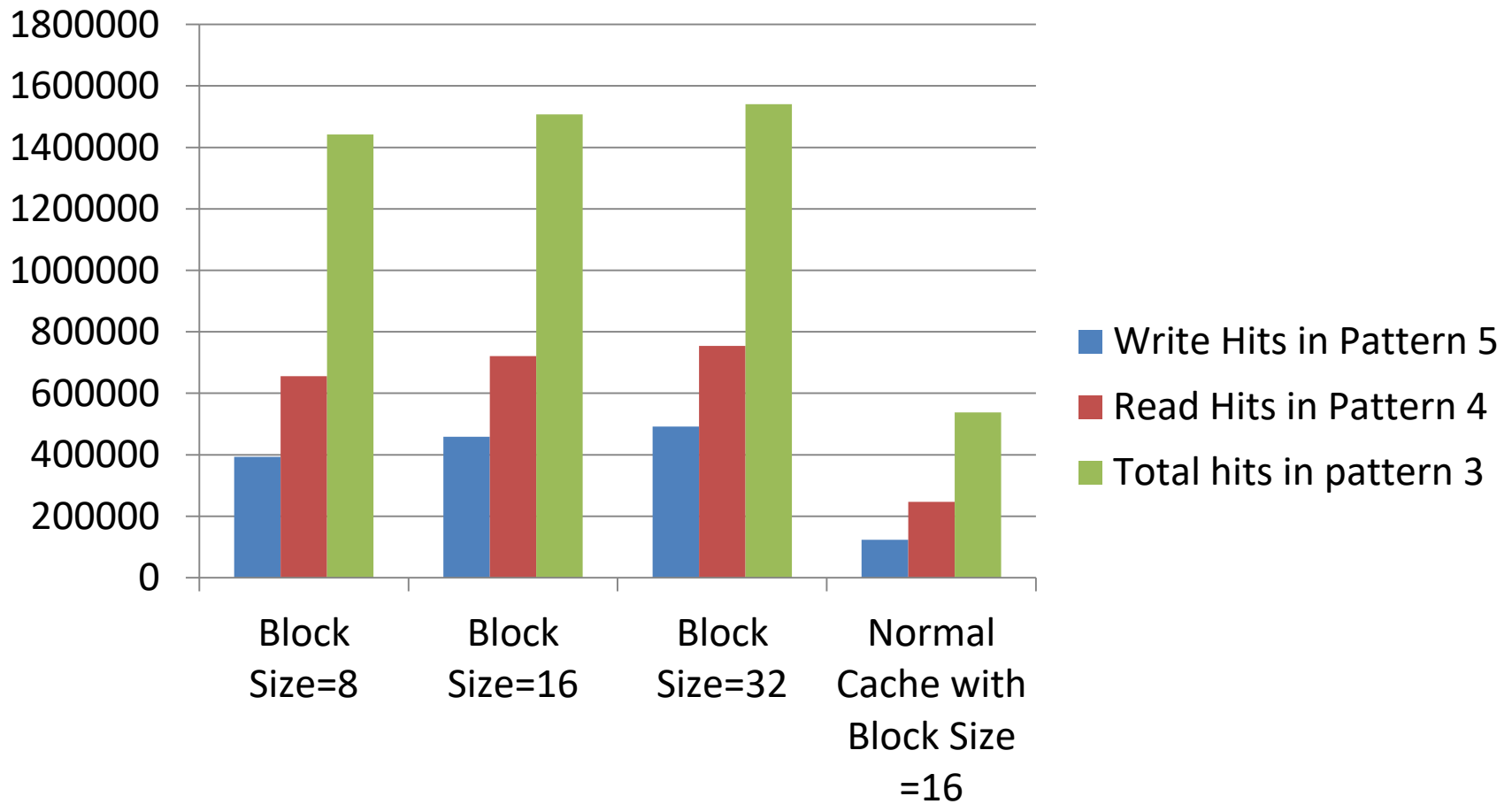
Image 1

This image was chosen to test the intuitive notion that checker boards have lot of similarity and thus an approximation cache like ours should perform quite well for this image.



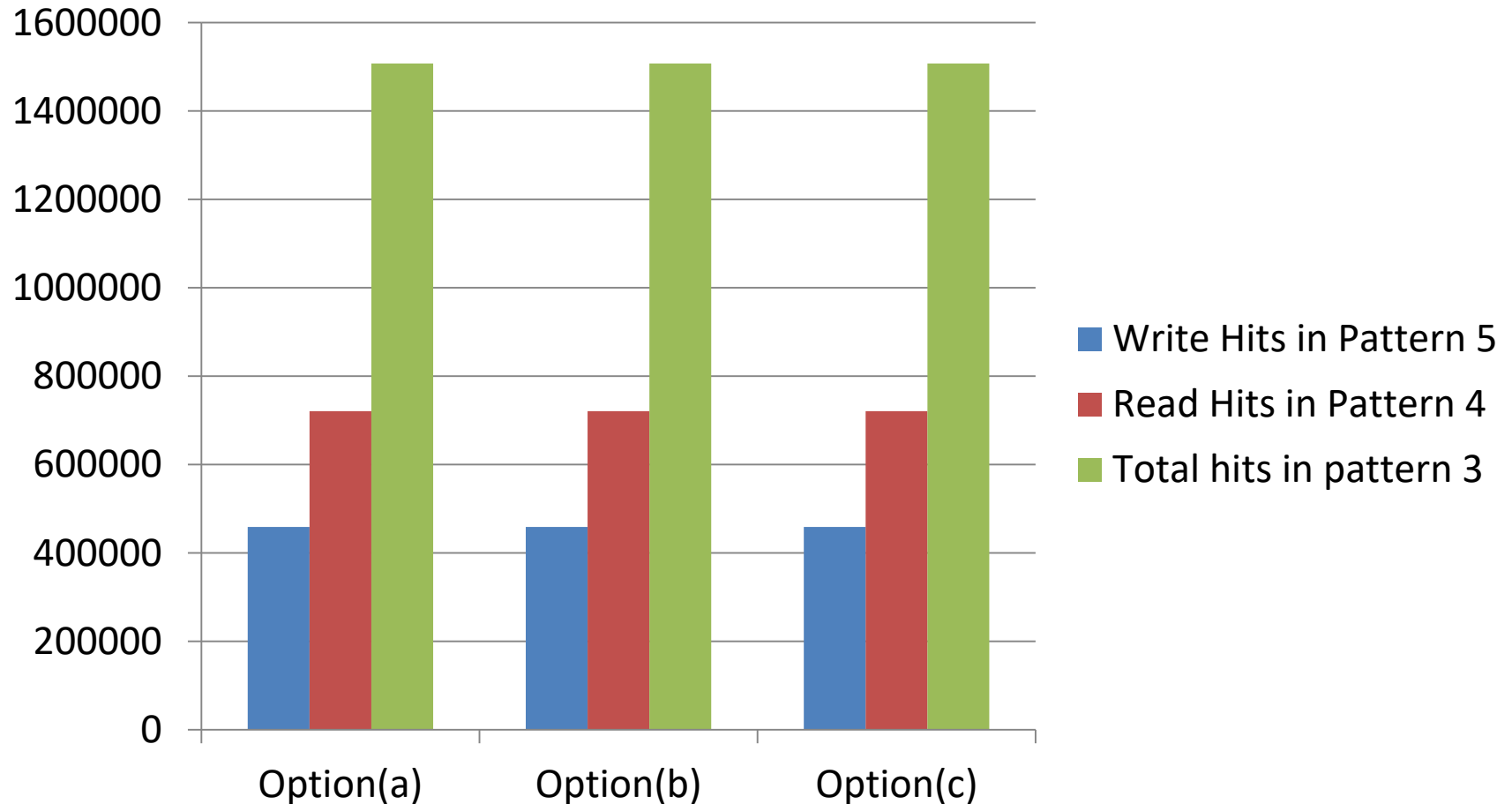
Ofcourse, though this image is an artificially manufactured image. It is a good starting point for verifying our hypotheses.

Hit Rate vs Block Size



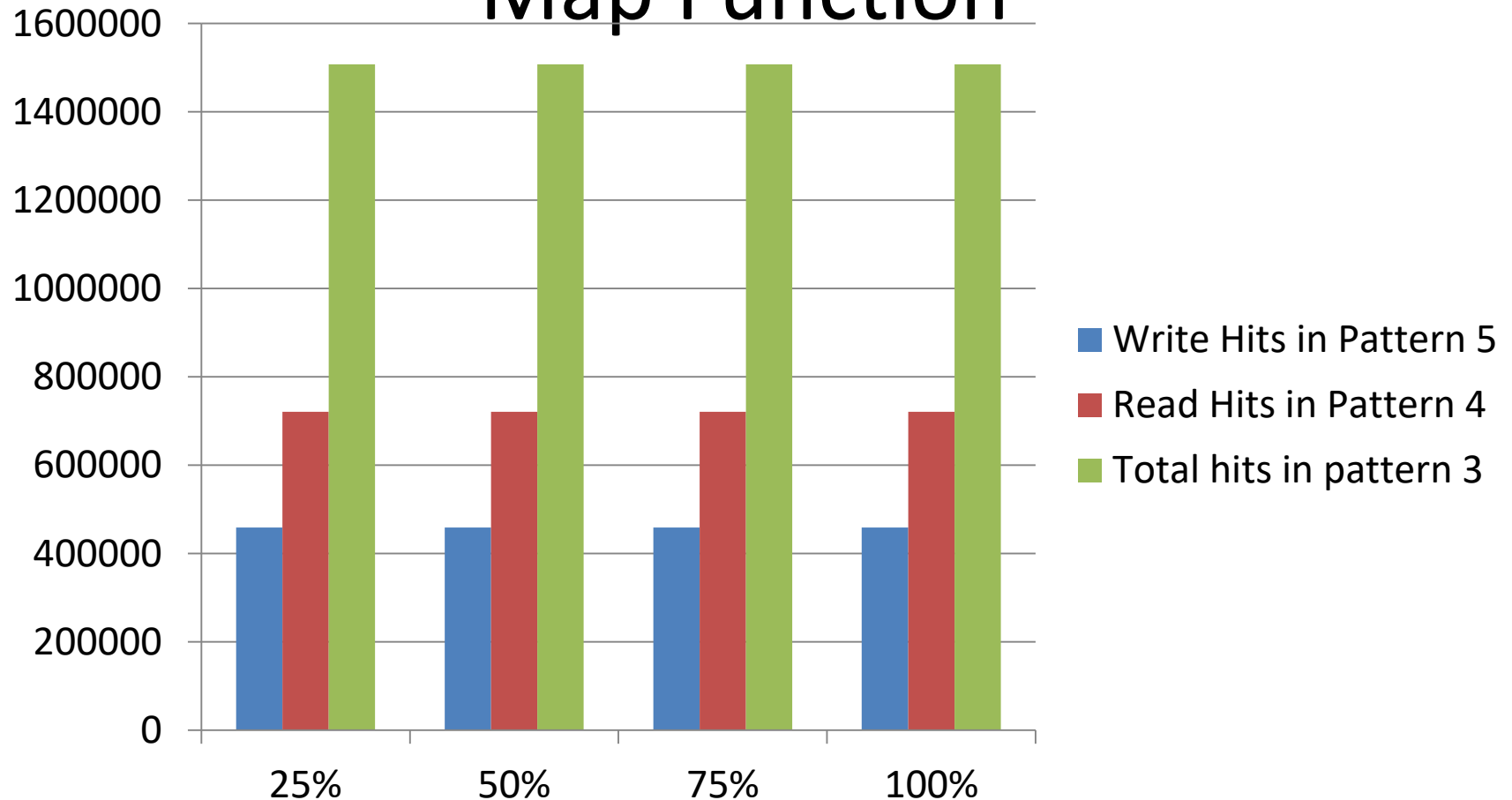
As expected the hit rate steadily increases in this case with the blockSize

Hit Rate vs Hash Function



Well you might ask why am I showing you 3 copies of the same bar graph? It is because the whole image has only two levels(0,0,0) and(255,255,255) and we don't have a hashFunction which can confuse them

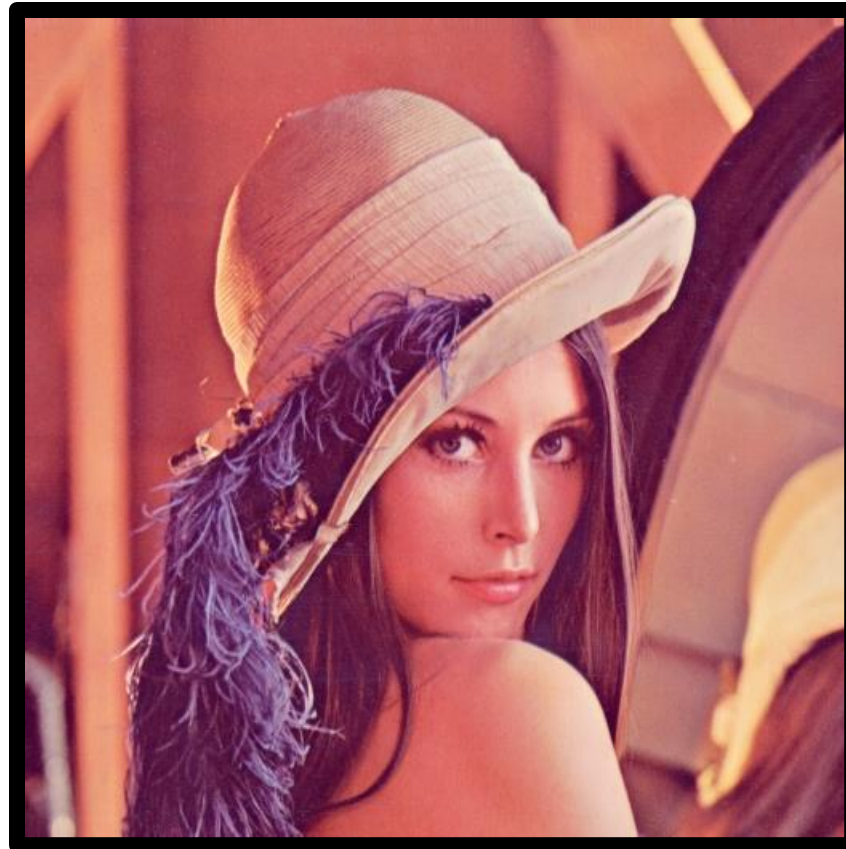
Hit Rate vs Information Retained in Map Function



Well you might ask why am I showing you 4 copies of the same bar graph? It is because even retaining 25% information about the hashFunction is enough to distinguish between (0,0,0) and (255,255,255)

Image 2

This image was chosen because apparently in the computer vision world, this image is used more than “Hello World” is used while teaching programming



This image also acts like one of the worst case scenarios for our approximation cache with nearly no similar areas and highly contrasting surfaces touching each other.

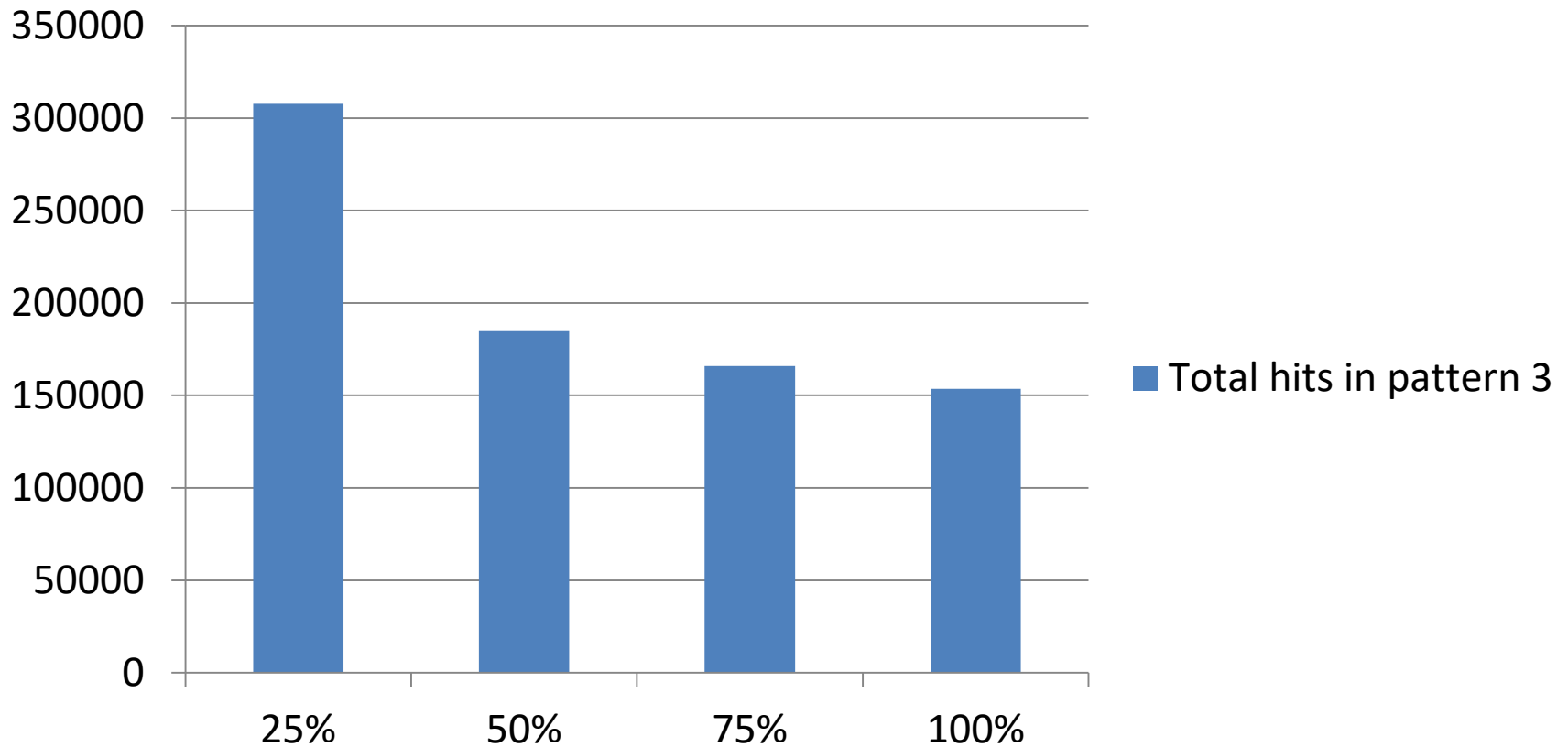
Cumulative Read Error

- Let $CRE=0$;
- We know every read returns a value to the top level.
- Let the value returned by the Doppleganger variant for a particular read be V_d and that by our normal cache be V_n then

$$CRE = \sum_{k=0}^{numReads} |V_{d,k} - V_{n,k}|$$

Cumulative Read Error vs Information Retained in Map Function

Total hits in pattern 3



For this image, I am showing the most significant information that we gain over the already known facts about the trends with respect to block size. Note there is still significant error due the hashing which is leading to loss of information

Image 3

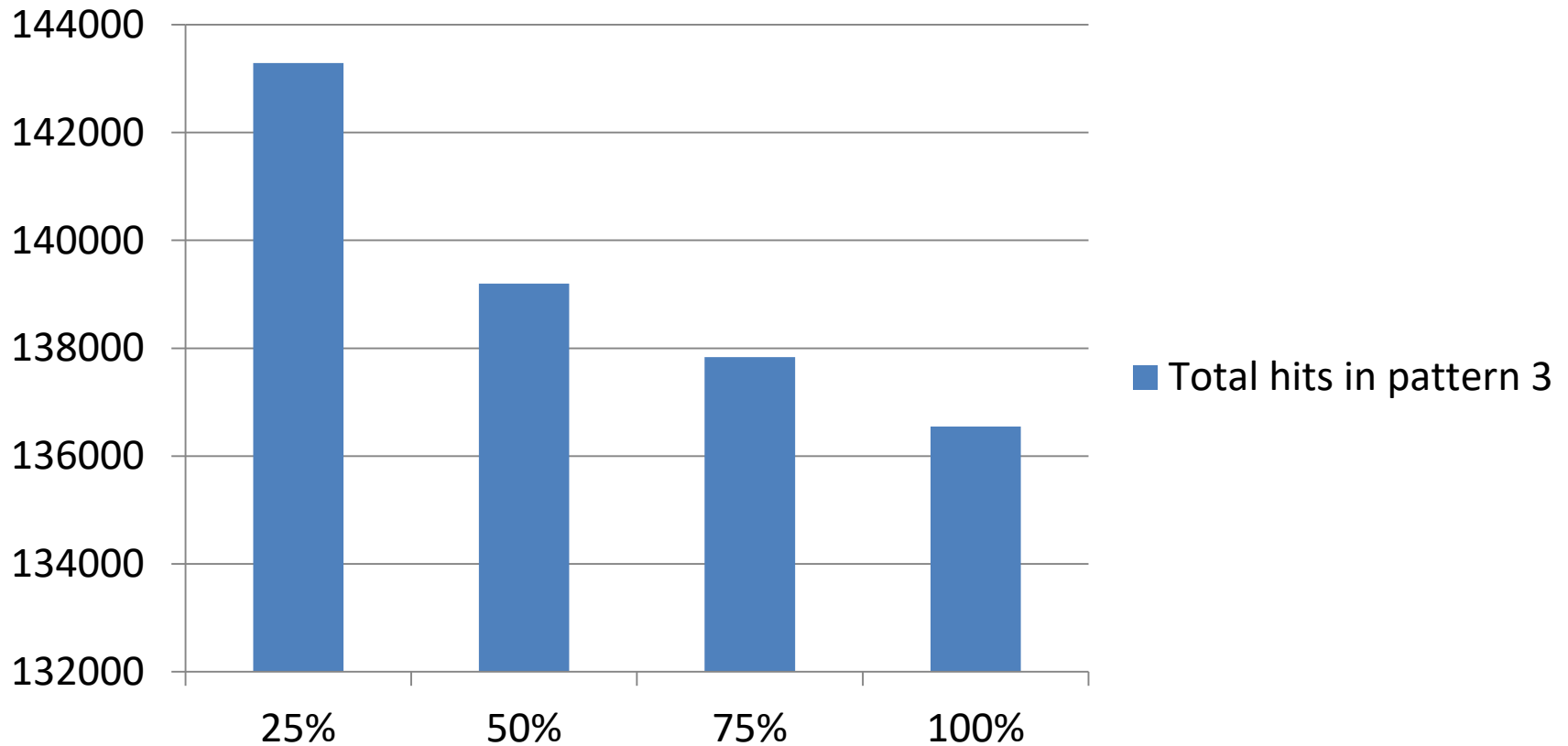
This image was chosen because this is a real world example of similarity in a image. The red foreground and black background give our cache multiple chances to cash in on these similarities



Note that this is also a dark image which means that the hash function chosen comes into play.

Cumulative Read Error vs Information Retained in Map Function

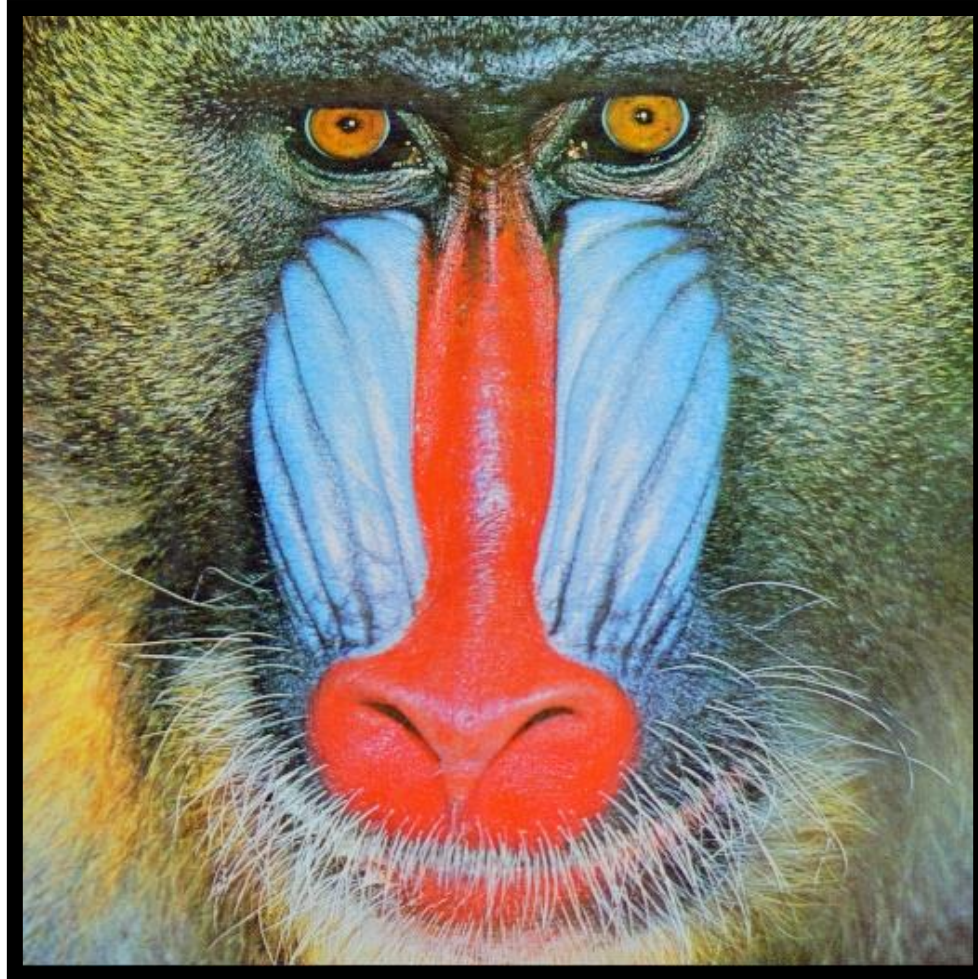
Total hits in pattern 3



Though the graph may look much more steeper than image 2, the information lost at each level significantly lesser than that of image 2. The dimensions of the graph are different with the least count on the y-axis being different

Image 4

This image was chosen because this is a real world example of how some channels can almost rule an area in an image, for e.g Red on the nose, Blue in the whiskers and cheeks



This is a good worst case scenario for Doppiegänger to test against since many images in real world are actually highly contrasting due to bad photography or the content itself being vivid

Image 5

This is an example of hypermodern art created by artificial means yet posing the same problems as real world images



With a whole wide world of animated and artificial art out there, this is just an example to see whether our cache system can cope up with this change

The Simulator

- Perhaps the most important development product out of this project would be the customizable high level LLC simulator for memory transactions.
- Some of its salient features are:
 - Customizable replacement policies independent for both data and tag arrays
 - High level memory transactions generator for graphic specific applications given an image.

The Simulator(contd.)

- Side modules for comparing the view as seen by the core(s) from the top
- High level LLC view customizable in the following factors:
 - Associativity
 - sizeOfBlock
 - physicalAddressSpace
 - cacheSize
 - dataArraySize
 - tagArraySize

The Simulator(contd.)

- Trace generator given a set of transactions and an input `physicalAddressSpace`
- Customizable hash and map functions for testing the efficacy of these in compressing the data.
- Modular code which makes it usable as a side module in a larger project

Questions + Demonstration



"That's all Folks!"