

# Cache Analysis and Utilization

*A Midterm Project Report Submitted  
in Partial Fulfillment of the Requirements  
for the Degree of*

Bachelor of Technology

By

Kaustubh chavan  
(112001017)



INDIAN INSTITUTE  
OF TECHNOLOGY  
**PALAKKAD**

**COMPUTER SCIENCE AND ENGINEERING  
INDIAN INSTITUTE OF TECHNOLOGY PALAKKAD**

# CERTIFICATE

*This is to certify that the work contained in project entitled “**Cache Analysis and Utilization**” is a bonafide work of **Kaustubh Chavan (RollNo. 112001017)**, carried out in the Department of Computer Science And Engineering, Indian Institute of Technology Palakkad under my guidance and that it has not been submitted elsewhere for a degree.*

**Dr. Unnikrishnan C**  
Assistant Professor  
Computer Science And Engineering  
unnikrishnan@iitpkd.ac.in  
Research Area: High Performance Computing,  
and Programming Languages

# **Acknowledgements**

I would like to express my gratitude to my mentor in this project, Prof. Unnikrishnan C, for giving me the opportunity to work on this project. The suggestions and encouragement provided by him helped me greatly in coordinating this project till where it's done and in writing this report itself. I am grateful to him.

# Contents

<b>Contents.....</b>	<b>4</b>
<b>Chapter 1.....</b>	<b>5</b>
<b>1. Introduction.....</b>	<b>5</b>
<b>Chapter 2.....</b>	<b>7</b>
<b>2.1 Existing Mapping Techniques.....</b>	<b>7</b>
2.1.1 Associative mapping.....	7
2.1.2 Direct Mapping:.....	7
2.1.3 Set Associative Mapping:.....	8
<b>2.2 Replacement Policy.....</b>	<b>9</b>
2.2.1 First In, First Out (FIFO):.....	9
2.2.2 Last In, First Out (LIFO):.....	9
2.2.3 Least Recently Used (LRU):.....	9
<b>2.3 Cache Miss Classification.....</b>	<b>10</b>
<b>Chapter 3.....</b>	<b>11</b>
<b>3.1 Enhancing Cache Performance: Architectural Approaches.....</b>	<b>11</b>
3.1.1 Bridging the Cache-Memory Speed Gap.....	11
3.1.2 Step 1: Reducing the Miss Rate using Victim Cache.....	12
<b>3.2 Implementation of Victim Cache:.....</b>	<b>12</b>
3.2.1 Step 2: Reducing the Penalty Ratio of Cache Miss.....	13
3.2.2 Step 3: Decreasing Memory Access Time.....	14
<b>Chapter 4.....</b>	<b>15</b>
<b>4.1 Optimization of cache memory for the implementation of algorithms for image processing.....</b>	<b>15</b>

<b>4.2 OPTIMIZING CACHE PERFORMANCE: A COMPREHENSIVE GUIDE TO CACHE HIERACHY AND PERFORMANCE METRICS.....</b>	<b>16</b>
<b>4.3 OPTIMIZATION OF VISION AND IMAGE PROCESSING ALGORITHMS.....</b>	<b>17</b>
<b>4.4 EXPERIMENTAL RESULTS.....</b>	<b>20</b>
4.4.1 Brightness Adjustment Experiment.....	20
4.4.2 Image Rotation and Tiling Optimization Experiment.....	24
4.4.3 Convolution Operation Experiment.....	26
<b>Chapter 5.....</b>	<b>28</b>
<b>5.1 Prior work.....</b>	<b>28</b>
<b>5.2 The objectives of this phase are multifold:.....</b>	<b>28</b>
<b>Chapter 6.....</b>	<b>29</b>
<b>6.1 Cache Prefetching for Image Processing.....</b>	<b>29</b>
<b>6.2 2D spatial locality in images.....</b>	<b>29</b>
<b>6.3. Cache prefetching techniques in image processing.....</b>	<b>30</b>
<b>6.4 EXPERIMENTS AND PERFORMANCE COMPARISON.....</b>	<b>31</b>
6.4.1 Testing Results for Image Dataset.....	37
6.4.2 Benchmarking Current Performance:.....	37
6.4.3 Integrating Cache Simulator:.....	37
6.4.4 Applying Prefetching Strategies:.....	37
6.4.5 Optimization and Performance Analysis.....	37
<b>Chapter 7.....</b>	<b>38</b>
<b>Conclusion and future work.....</b>	<b>38</b>
<b>6. References.....</b>	<b>38</b>

# Chapter 1

## 1. Introduction

Cache memory is a crucial component in computer systems due to its incredibly fast access times, which are unmatched by other types of memory and have a profound impact on program execution. It operates on the principle of the 'locality of reference,' where most memory accesses are confined to specific, localized areas within the memory. By storing active instructions and frequently accessed data, cache memory significantly reduces the average memory access time, thereby enhancing program performance.

Cache memory is organized into multiple levels, ranging from the smallest, fastest L1 cache integrated within the processor to larger, slower L2 and L3 caches located on separate chips. In multi-core processors, each core often has its own dedicated L1 cache, with the last level of cache being shared among all cores. [\[1\]](#)

When the CPU needs to access data, it first checks its primary cache. If the data is found there, it results in a 'hit'; otherwise, a 'miss' occurs, and the data is retrieved from the main memory. The hit ratio, the ratio of hits to the total number of hits and misses, plays a crucial role in cache performance. Misses can happen for various reasons, such as the word being accessed for the first time, inadequate cache size, or the cache's limited capacity.

The cache miss rate and the time required to handle cache misses are two critical factors influencing cache performance. It is possible to lower cache miss rates with tools such as victim caches (which store evicted cache lines temporarily) and column-associative caches.

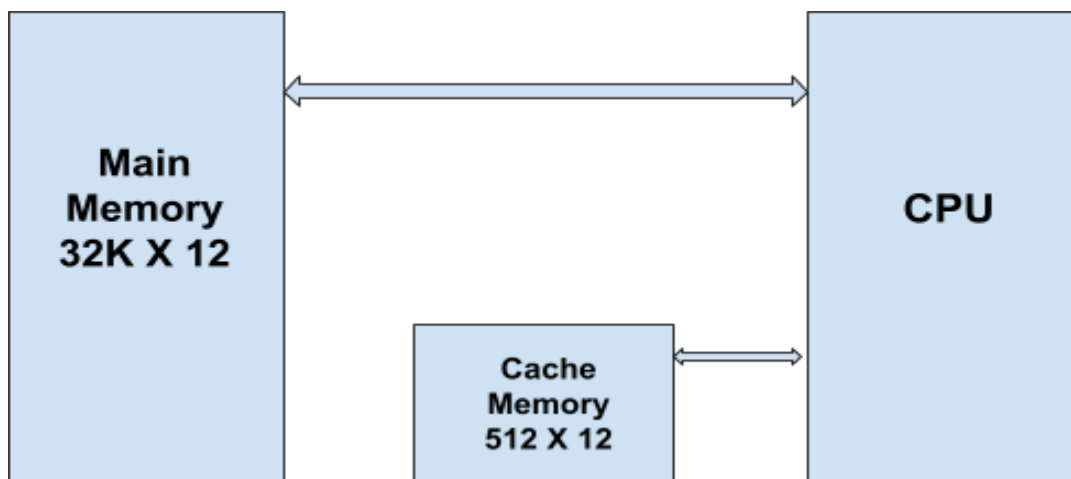


Figure 1. Location of cache memory

Cache memory serves as a vital intermediary between the CPU and slower main memory, facilitating fast data access. Its proximity to the processor makes it essential for data synchronization. Although cache memory offers speed advantages over main memory and RAM, it consumes more energy due to its on-chip location. CPU[1]

As processor speeds continue to increase, there is a pressing need to further enhance cache memory speed. In this report, we explore three enhancement techniques—**victim cache**, **sub-blocks**, and **memory banking**—to improve cache speed at each level. We evaluate the performance of cache memory by considering variables such as miss penalty ratio, cache access speed, and miss rate ratio after implementing these techniques.

In essence, cache memory acts as a bridge between the processor and main memory, making data retrieval faster and more efficient. The ongoing evolution of cache systems plays a pivotal role in shaping the performance of multi-core systems and narrowing the gap between processor and memory speeds. This report will delve into these improvements and their potential to enhance cache memory's speed and performance relative to main memory."

## Chapter 2

### 2.1 Existing Mapping Techniques

Mapping techniques in cache organization are crucial in optimizing the efficiency of cache memory. Here, we delve into the three primary mapping techniques.

#### 2.1.1 Associative mapping

Cache memory operates based on the principle of associative mapping, recognized for its flexibility and speed. This method employs associative memory to store memory addresses and their associated data, as depicted in Figure 2. Each address in the cache can correspond to any word in the main memory. First, the computer looks at a 15-bit address in the argument register. It checks if this address is in the associative memory. If it is, it quickly gets the 12-bit data for the CPU. But if the address isn't in the associative memory, it sends a request to the main memory to find the matching address. Upon a successful match, the associated data pair is retrieved and stored in the associative memory.

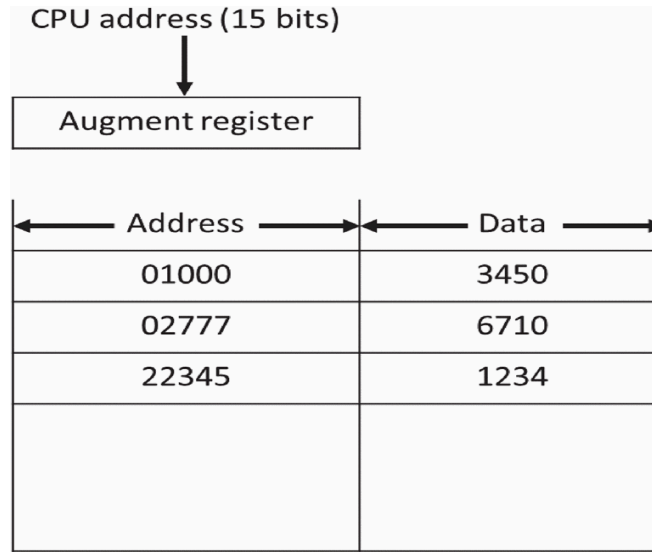


Figure 2. Associative

mapping(<https://www.semanticscholar.org/paper/Cache-Memory%3A-An-Analysis-on-Performance-Issues-Sonia-Alsharef/4abb6f1b69d6ec69606e8a11ba9c27807540ded8/figure/1> )

### 2.1.2 Direct Mapping:

Figure 3 shows how to use RAM as a cache. The processor's address is split into two parts: the index and the tag. The index is made up of the 9 least significant bits of the address, and the tag is made up of the 6 most significant bits. Both the index and the tag must be included in the main memory address. The cache is typically  $2^k$  words in size, and the main memory is  $2^n$  words in size. The  $n$ -bit main memory address is divided into a  $K$ -bit index field and an  $NK$ -bit tag field. The direct mapping of cache addresses uses the  $n$ -bit main memory address to access memory and the  $K$ -bit index address to access the cache. Words in the cache are organized as shown in Figure 3. Each word consists of both data and its associated tag. When a new word is fetched into the cache, the cache bits are stored along with the data bits. If the CPU requests a word from memory, the cache is accessed using its address, using the index field. The tag part of the processor address is then compared to the tag field of the cache word at that index. If there is a match, the required data is retrieved from the cache.



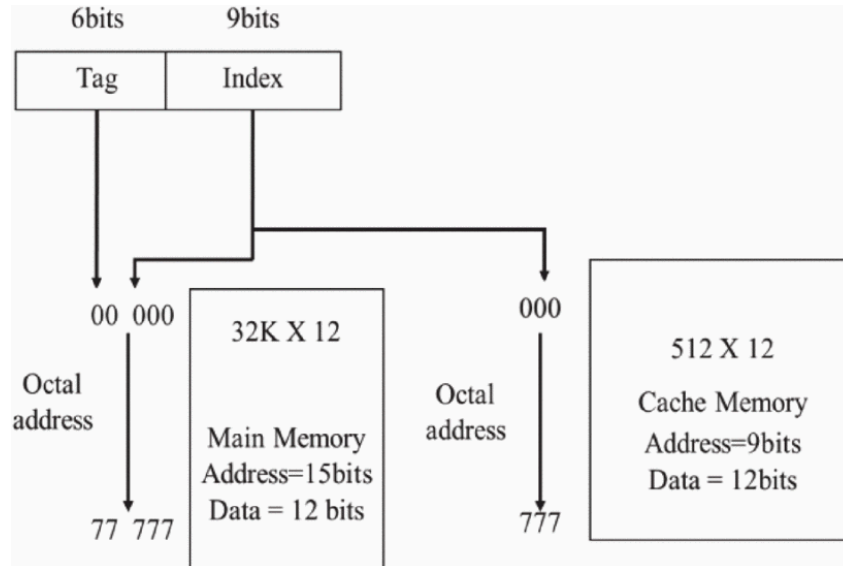


Figure 3: DirectMapping

<https://www.semanticscholar.org/paper/Cache-Memory%3A-An-Analysis-on-Performance-Issues-Sonia-Alsharef/4abb6f1b69d6ec69606e8a11ba9c27807540ded8/figure/2> )

### 2.1.3 Set Associative Mapping:

Set associative mapping builds upon direct mapping to enhance cache performance by allowing multiple words to share the same index address. Each word is stored alongside its tag, and multiple tag items are considered a set within one cache word. For example, consider a main memory with 1024 words, and each cache word containing 2 data words. A set-associative cache of a specified set size, as shown in Figure 4, can consider K words of main memory for each cache word. When a processor initiates a memory request, the cache is accessed using the address's index value. The tag field of the processor address is then compared to the combined tags in the cache memory to determine if there is a match. This logical comparison is performed by associatively searching tags within the set, similar to associative memory searches. The increased set size improves the hit ratio, as different tags can be allocated to the same index. Hence, it's called 'set associative.'

Index	Tag	Data		Tag	Data
000	01	3450		02	5670
777	02	6710		00	2340

Figure 4: Set associative mapping cache

(<https://www.semanticscholar.org/paper/Cache-Memory%3A-An-Analysis-on-Performance-Issues-Sonia-Alsharef/4abb6f1b69d6ec69606e8a11ba9c27807540ded8/figure/3>)

## 2.2 Replacement Policy

Replacement policies are critical in managing data eviction and replacement in set-associative and fully-associative caches. The choice of policy can significantly impact performance, and the selection should align with the specific characteristics of the application. Common options encompass:

### 2.2.1 First In, First Out (FIFO):

This policy ejects the oldest line that was introduced into the set or cache.

### 2.2.2 Last In, First Out (LIFO):

It prioritizes the removal of the most recently introduced line within the set or cache.

### 2.2.3 Least Recently Used (LRU):

This policy targets the least recently accessed line in the set or cache for eviction.

Random: The random policy selects a line for eviction at random from the set or cache.

In certain microprocessors, the replacement policy can be adapted by software to match the expected reference pattern. For example, the Cell microprocessor offers programmers the ability to configure a replacement management table. This table, organized according to address space ranges, identifies potential candidates for replacement. In practice, this allows different parts of an application to choose distinct replacement policies. For

instance, a streaming application might opt for LIFO replacement for its cache lines, while a critical interrupt handler could select LRU or even pin-specific cache lines by specifying an empty replacement list. This flexibility enables the optimization of cache performance according to the unique demands of different software components.

## 2.3 Cache Miss Classification

When analyzing the factors that impact cache performance, it is beneficial to categorize different types of cache misses according to their root causes, whether they are real or perceived. This category provides valuable insights into potential approaches for improving cache performance. Hill and Smith (1989) introduced an easy-to-understand framework consisting of three separate categories, commonly known as the three C's:

**Conflict** misses refer to misses that occur in a set-associative cache but are not present in a fully-associative cache of the same size.

**Capacity** misses are an issue that arises in fully-associative caches with restricted sizes. It is important to note that such misses would not be a concern in caches with infinite capacity.

**Compulsory** Cache Misses: These are cache misses that would occur regardless of the cache size being indefinitely large.

In order to provide comprehensive coverage, an additional category known as "Coherence Misses" has been introduced inside the structure of cache-coherent multiprocessor systems. The lack of coherence arises as a consequence of the execution of cache coherency protocols. However, due to the primary emphasis of this work on uniprocessor systems, the fourth category remains unexplored in further detail.

In regard to the core cache parameters previously described, several general trends emerge.

Increasing the cache size decreases the occurrence of capacity misses, but this comes at the cost of more complicated cache logic and potential latency increases.

The increase in cache associativity has the effect of reducing the occurrence of conflict misses. However, this enhancement also adds more complex cache logic, which may result in increased latency.

Increasing the cache line size has the potential to reduce the occurrence of obligatory misses, while it may also lead to an increase in the frequency of conflict misses.

The method of prefetching, which will be further examined in future discussion, has the potential to reduce the frequency of obligatory misses.

The overall pattern suggests that caches with greater capacities and stronger associations are likely to result in increased hit rates. However, a tradeoff arises when considering cache capacity, associativity, and cache latency, requiring a careful balance to ensure the timely provision of data to the processor. [\[2\]](#)

## **Chapter 3**

### **3.1 Enhancing Cache Performance: Architectural Approaches**

#### **3.1.1 Bridging the Cache-Memory Speed Gap**

This section introduces an architectural solution to bridge the speed gap between cache memory and main memory. In this innovative architecture, a combination of techniques is integrated sequentially, systematically addressing the limitations of each one. This step-by-step approach is designed to comprehensively enhance cache memory speed, with each technique contributing to improved memory performance. In this report, various techniques and methodologies are examined to evaluate the effectiveness of this proposed technology. The architecture is initially presented as a flowchart in Figure 5 represents the proposed architecture for improving cache performance. This flowchart outlines the key steps involved in enhancing cache performance, including the use of victim caches, sub-blocks, and memory banking.

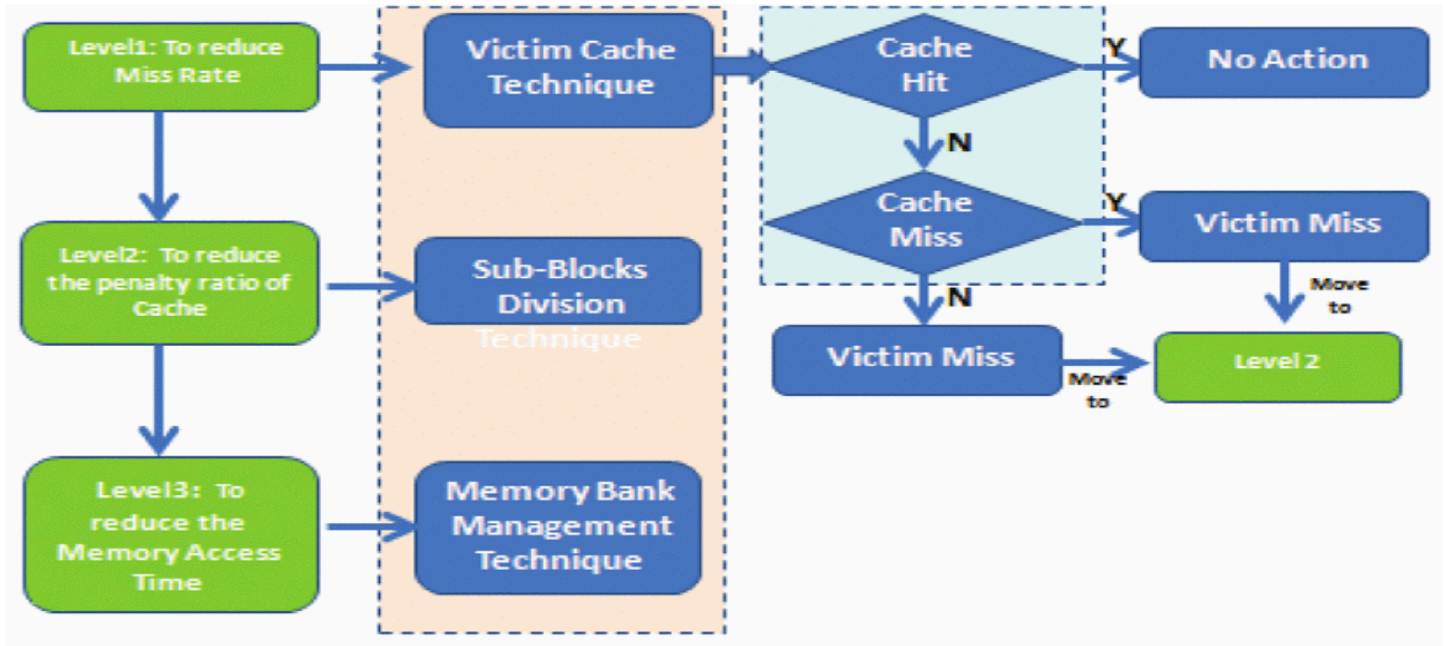


Figure 5. Enhancing Cache Performance: Architectural Approaches

(<https://www.semanticscholar.org/paper/Cache-Memory%3A-An-Analysis-on-Performance-Issues-Sonia-Alsharef/4abb6f1b69d6ec69606e8a11ba9c27807540ded8/figure/4> )

### 3.1.2 Step 1: Reducing the Miss Rate using Victim Cache

A victim cache, sometimes called hardware cache memory, is a valuable tool used to minimize delays in direct-mapped cache systems and decrease instances of conflicting data misses. This cache is strategically placed along the path where the Level 1 cache gets refilled. It works by catching cache leaks and keeping them on standby. So, whenever data is pushed out of the Level 1 cache, it is simultaneously stored in the victim cache. The victim cache springs into action only when there's a miss in Level 1. In the event of a hit, it results in the exchange of the matching victim cache line and the Level 1 cache line.

## 3.2 Implementation of Victim Cache:

When we have a **cache hit**, there's nothing special that needs to be done. In a **Cache Miss** scenario where the Victim Cache comes to the rescue (**Victim Hit**), we do a little swap dance between the blocks in both the victim cache and the main cache. As a result, the block that was most recently placed in the victim cache now becomes the most recently used. However, when we have a **cache miss** and the victim cache can't help (**the victim miss**), we go a step further. We bring in the missing part from the next level of the cache into the main cache.

Meanwhile, the outgoing data from the main cache finds a temporary home in the victim cache. Imagine a direct-mapped cache that has two blocks, A and B, sharing the same slot. Now, connect it to a second-level fully associative victim cache that contains blocks C and D. This whole process unfolds as illustrated in Figure 6, creating a back-and-forth pattern between A and B. So, when a victim cache hit occurs, parts of blocks A and B get swapped around, and the Least Recently Used (LRU) block in the victim cache stays put. It's almost like giving the direct-mapped L1 cache a taste of associativity, which helps cut down on the number of conflicting misses. In a system with two cache levels, Level 1 and Level 2, and a particular policy that keeps them from storing the same data, Level 2 acts as a sort of safety net, coming to the aid of Level 1.

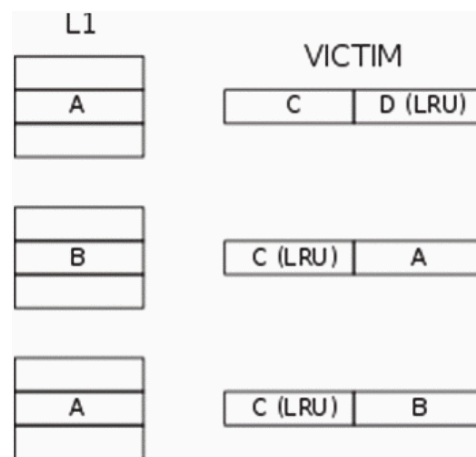


Figure 6. Implementation example

(<https://www.semanticscholar.org/paper/Cache-Memory%3A-An-Analysis-on-Performance-Issues-Sonia-Alsharef/4abb6f1b69d6ec69606e8a11ba9c27807540ded8/figure/5> )

### 3.2.1 Step 2: Reducing the Penalty Ratio of Cache Miss

Cache performance can suffer when we use tags that need a lot of space because it slows down the cache. To deal with this, we use bigger blocks. While this helps reduce the number of times we miss the cache, it increases the time it takes to fetch data when a miss happens because we have to move the entire block between the cache and other parts of the memory. To solve this problem, we divide each block into smaller parts, called sub-blocks, as you can see in Figure 7. Each sub-block has a little piece of information called a valid bit. Even though the tag stays valid for the whole block, we only have to read a single sub-block when there's a miss. This makes the miss penalty much smaller because we're not moving a whole block between the cache and memory every time.

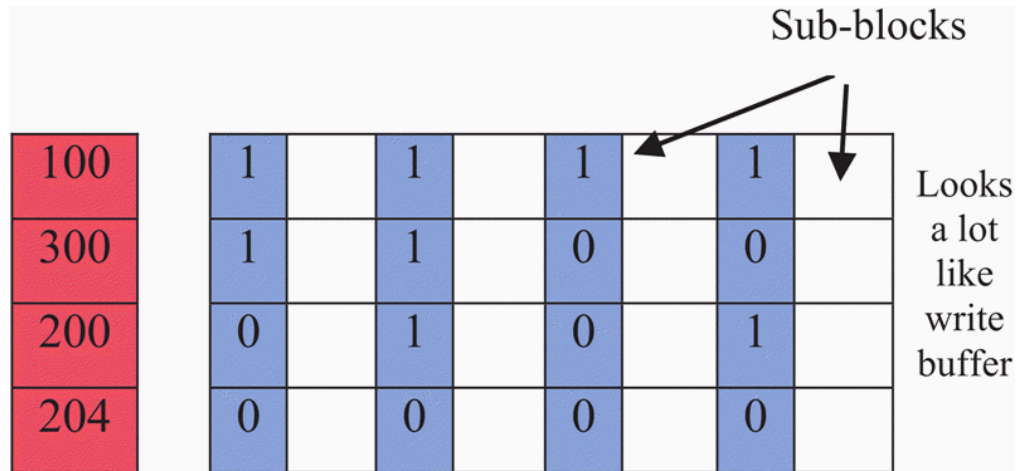


Figure 7. Block division into sub-blocks

(<https://www.semanticscholar.org/paper/Cache-Memory%3A-An-Analysis-on-Performance-Issues-Sonia-Alsharef/4abb6f1b69d6ec69606e8a11ba9c27807540ded8/figure/6> )

### 3.2.2 Step 3: Decreasing Memory Access Time

Memory banks, in the realm of computer architecture, serve as discrete logical units tied to specific memory slots. The memory controller, a critical component in computer systems, Coordinates these memory banks. Each bank is associated with a particular memory slot, and this association is essential for memory organization. When data, represented as items such as  $a(n)$ , is stored in one bank, say bank (b), the subsequent item,  $a(n+1)$ , is placed in the next consecutive bank, namely bank (b+1). This sequential arrangement ensures efficient data storage and retrieval. To optimize the handling of data and reduce potential delays caused by memory access times, cache memory is partitioned into multiple banks. This strategy enables concurrent data retrieval and storage operations, enhancing overall system performance. The number of memory modules required aligns with the number of data bits in the bus, a critical consideration in memory design. [1]

A memory bank can consist of numerous memory modules, as demonstrated in Figure 8.

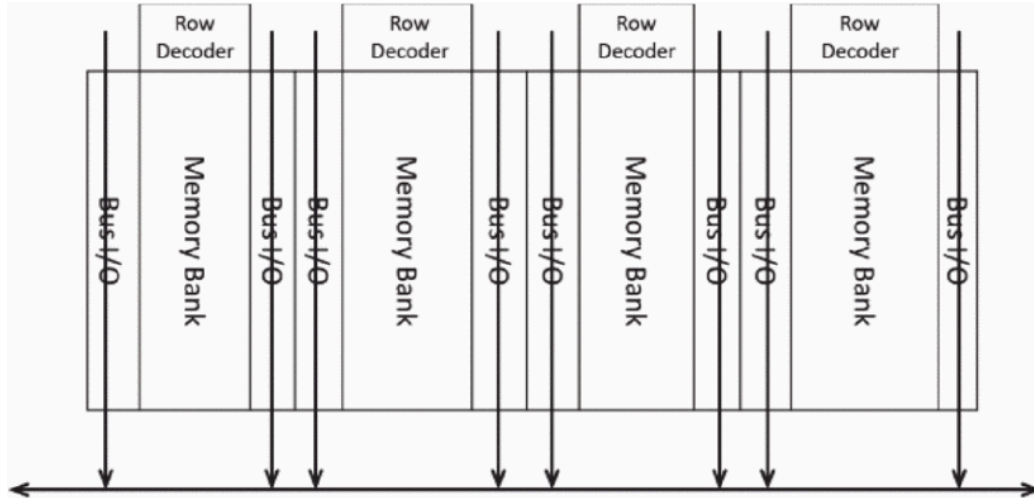


Figure 8. Memory bank modules

(<https://www.semanticscholar.org/paper/Cache-Memory%3A-An-Analysis-on-Performance-Issues-Sonia-Alsharef/4abb6f1b69d6ec69606e8a11ba9c27807540ded8/figure/7>)

## Chapter 4

### 4.1 Optimization of cache memory for the implementation of algorithms for image processing

Processors can process data much faster than memory can supply it, leading to a significant speed impact due to main memory latency. This bottleneck can be alleviated by effectively utilizing cache memory, which operates between 10 and 100 times faster than main memory. However, challenges arise when working with large data sets, such as those in image processing and computer vision algorithms that require extensive memory buffers.

New high-resolution cameras and real-time video streams, for instance, demand memory buffers exceeding 100 megabytes per second. While cache memory size is limited, ranging from about 128 KB for L1 to 4–16 MB for LLC, exploiting the locality of reference can enhance cache memory's hit ratio, thus improving overall system performance. [3]

The locality of reference, which is categorized into temporal and spatial localities, plays a crucial role. Temporal locality suggests that recently referenced items are likely to be referenced again soon, while spatial locality involves using data elements stored close to the currently referenced item. Regular access patterns to memory locations increase the locality of reference and system performance.



Software developers, despite lacking direct access to hardware-controlled cache memory, can inadvertently create inefficient vision systems by neglecting cache-related issues in their code. The preferred approach involves indirect control of cache memory through optimization techniques applied at both the algorithmic and implementation levels. Profiling tools aid in obtaining measurements and analyzing time hot spots, guiding optimizations based on a deep understanding of hardware architecture, memory operation, image format, and algorithm nature.

The focus of this work is on image processing and computer vision systems, where optimization primarily targets data rather than instruction profiling and optimization. By enhancing cache memory, the report presents optimizations for image processing and computer vision algorithms. Profiling tools identify bottlenecks and time hot spots, with a comparison between direct algorithm implementation and optimized algorithms showcasing performance improvements.

## 4.2 OPTIMIZING CACHE PERFORMANCE: A COMPREHENSIVE GUIDE TO CACHE HIERARCHY AND PERFORMANCE METRICS

Various cache memory architectures exist, typically comprising three or four levels, as depicted in Fig. 9. In multi-core processors, L1 and L2 caches are specific to each core, while L3, or Last Level Cache (LLC), is commonly shared among all cores. L1 cache, the fastest and closest to the CPU after registers, has a latency of 4-5 cycles and a size of up to 256KB, though more powerful processors may feature a 1MB L1 cache. The L1 cache includes two types: L1 D for data and L1 I for instructions. L2 cache, larger than L1, stores more data with a slightly slower access time (around 7 cycles) and is used for both instructions and data. [\[3\]](#)

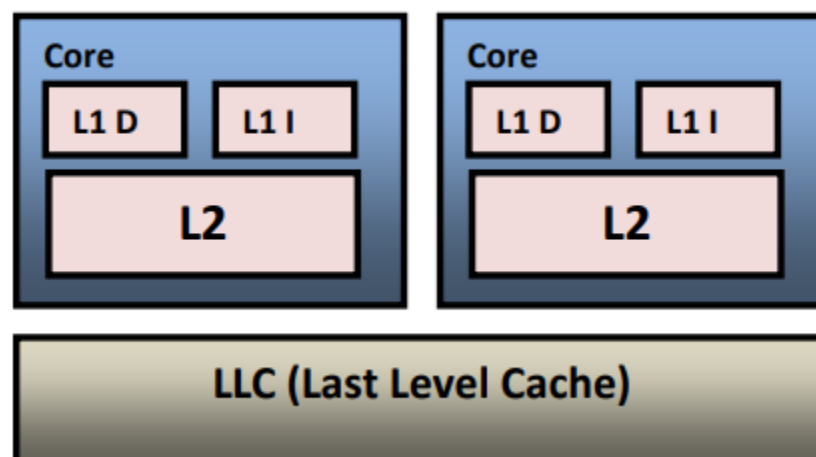


Fig. 9 illustrates L1, L2, and LLC cache memories.  
([https://jaes.journals.ekb.eg/article\\_87899\\_8dd6262bcebd0c80b8a8f9e14c40dff8.pdf](https://jaes.journals.ekb.eg/article_87899_8dd6262bcebd0c80b8a8f9e14c40dff8.pdf))

LLC is the largest and slowest shared memory for all cores, ranging from 2–8 MB in size and having a latency of 20–30 cycles. Since cache memory can't store extensive data, a portion is kept in the cache, and the rest remains in main memory. When the processor requests a memory location (read or write), it checks the cache for a corresponding entry. If found, a cache hit occurs; otherwise, a cache miss occurs, and new data is copied from main memory to the cache. Data and instructions move between main memory and cache in blocks called cache lines.

Cache controllers aim to predict future data access by loading additional data during the replacement process. Spatial locality retrieves data from neighboring addresses, while temporal locality involves reusing specific data within a short time duration. Cache memory performance is assessed using various metrics, with the primary goal of reducing program execution time. Profiling tools, such as Intel VTune Profiler, capture cache memory events. The LLC Miss Count number is used to find the hit ratio, which shows the percentage of visits that were successful in the cache. Memory bound is another measure that shows how many cycles were wasted waiting for data or instructions to be loaded or stored. Metrics like L1 Bound, L2 Bound, and L3 Bound give you more information about what causes delay. Tools that profile hardware events and look at time help improve speed and cache locality by making algorithms work better. [3]

## **4.3 OPTIMIZATION OF VISION AND IMAGE PROCESSING ALGORITHMS**

You can store multidimensional arrays in linear storage using either row-major order or column-major order. The selection of either row-major or column-major order in the context of picture pixels affects the arrangement of adjacent pixels. Figure 10 depicts an example image that is stored in memory using row-major order. In this arrangement, pixels within a row are stored consecutively in nearby memory regions, whereas pixels within a column are placed further apart. Column-major order displays a pattern that is inverted. [3]

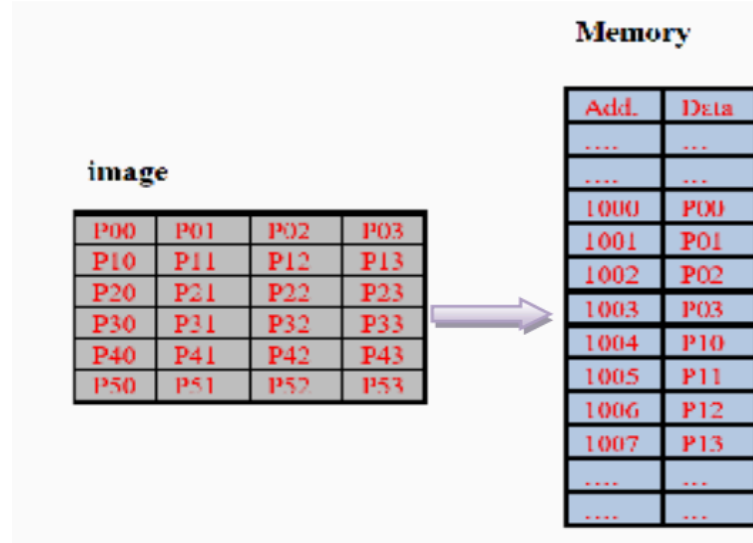


Figure 10. Row-Major order  
[https://jaes.journals.ekb.eg/article\\_87899\\_8dd6262bcebd0c80b8a8f9e14c40dff8.pdf](https://jaes.journals.ekb.eg/article_87899_8dd6262bcebd0c80b8a8f9e14c40dff8.pdf)

Optimizing simple image scan algorithms involves considering the major order used for image representation. Adjusting the inner and outer loops in the algorithm can enhance spatial locality, taking into account cache memory issues, thus improving performance and reducing processing time for various image operations. However, not all operations have straightforward optimization solutions. Complex operations and algorithms require thoughtful analysis, iterations, time evaluation, and the use of profiling tools.

For example, an image rotation method involves extracting data from one memory buffer and putting it to another buffer in a different order. Enhancing the order in which reading is done might result in ineffective locality for writing, and vice versa. Figure 11 demonstrates the application of picture tiling, a technique that partitions the image into smaller sections for performing operations. Partitioning data into smaller segments enhances the probability of cache hits, dependent upon variables such as cache capacity, kind of operation, and size of the picture. [3]

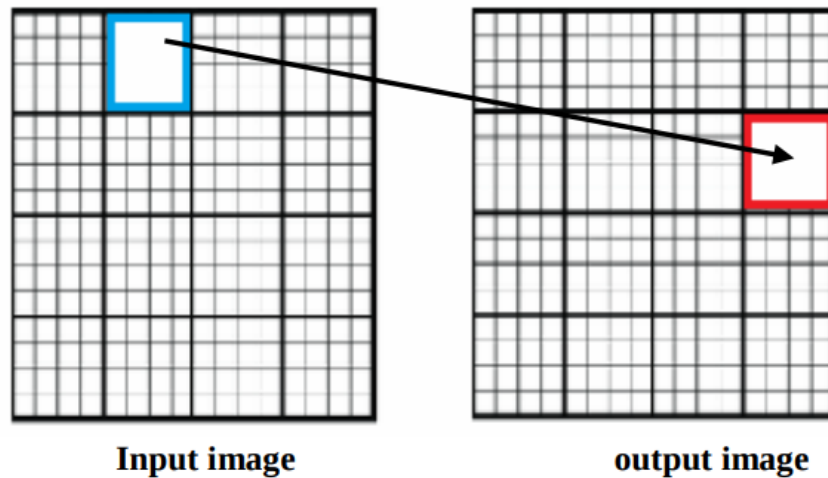


Figure 11: Image Tiling  
([https://jaes.journals.ekb.eg/article\\_87899\\_8dd6262bcebd0c80b8a8f9e14c40dff8.pdf](https://jaes.journals.ekb.eg/article_87899_8dd6262bcebd0c80b8a8f9e14c40dff8.pdf))

Figure 12 shows what happens when you use more than one convolutional kernel for convolution and correlation operations, which are common in image processing and computer vision. Large kernel sizes or a high number of kernels can increase the cache miss ratio and decrease temporal locality.

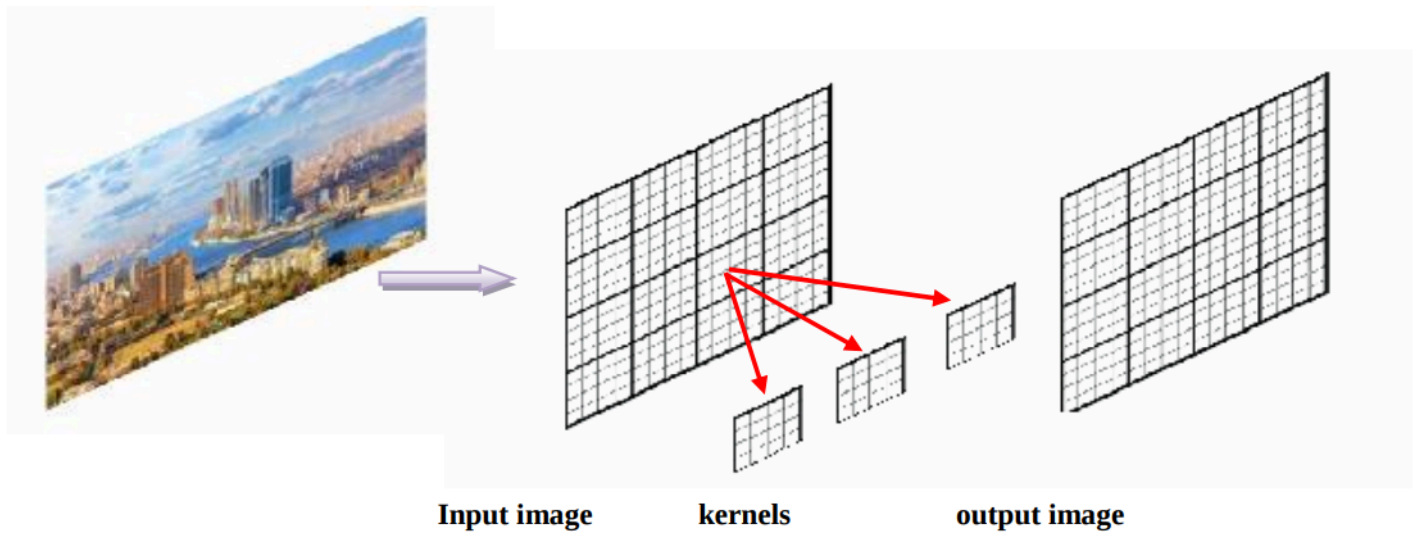


Figure 12. Multiple convolutional kernels on temporal locality([https://jaes.journals.ekb.eg/article\\_87899\\_8dd6262bcebd0c80b8a8f9e14c40dff8.pdf](https://jaes.journals.ekb.eg/article_87899_8dd6262bcebd0c80b8a8f9e14c40dff8.pdf))

Applications like simple edge detection or image smoothing often require smaller convolution kernels, while tasks like object tracking or pattern recognition may demand larger kernels. Convolutional Neural Networks (CNNs) extensively use multiple kernels, and optimizing their storage in cache memory can enhance temporal

locality. Placing the kernel loop as an outer loop enhances the temporal locality of the kernel data. However, this arrangement can potentially diminish the spatial locality of the image data.

Optimizing algorithms requires consideration of profiling tools, hardware events, and other metrics to design and implement high-performance systems by increasing both temporal and spatial locality.

## 4.4 EXPERIMENTAL RESULTS

In this section, we delve into the comprehensive analysis of our experimental results. Our primary focus is on testing and scrutinizing various image processing and computer vision algorithms, utilizing diverse images of varying sizes to investigate the influence of cache memory capacity. To gather detailed insights, we are using the Intel VTune Profiler software, which facilitates the collection of diverse hardware events and memory metrics, aiding in code analysis, hotspot detection, and statistical evaluations.

### 4.4.1 Brightness Adjustment Experiment

For our initial experiment, we applied a basic modification to the image's brightness, employing two distinct codes (Code-1 and Code-2). The image was stored in memory using row-major order, prompting us to explore the impact of utilizing the vertical index as the outer loop on spatial locality.



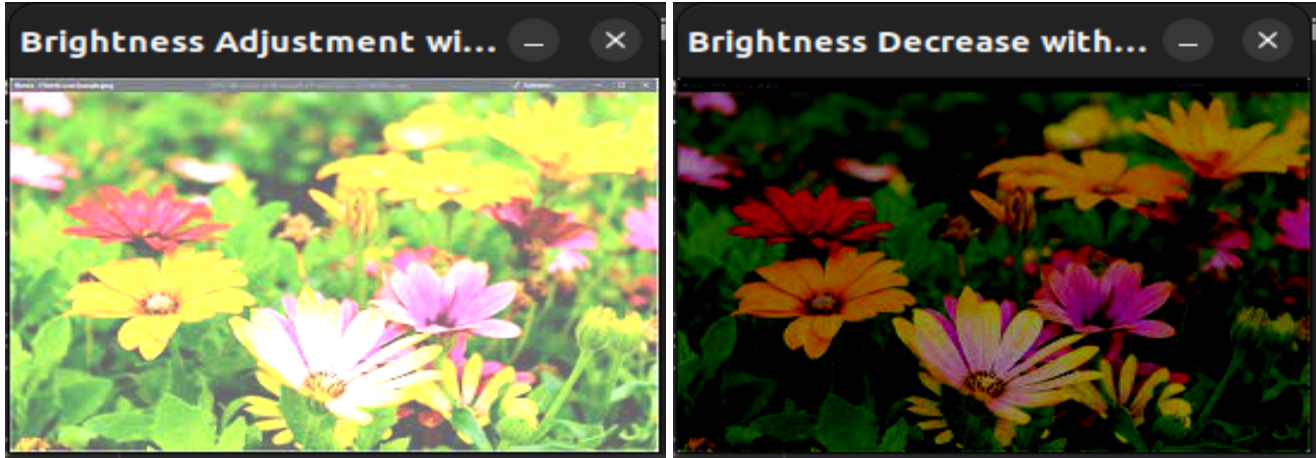


Figure 13. Brightness Transformation

#### Code 1


```
for (int x = 0; x < width; ++x) {
    for (int y = 0; y < height; ++y) {
        for (int c = 0; c < inputImage.channels(); ++c) {
            outputImage.at<cv::Vec3b>(y, x)[c] = cv::saturate_cast<uchar>(inputImage.at<cv::Vec3b>(y, x)[c] -
adjustment);
        }
    }
}
```


#### Code 2

```
for (int y = 0; y < height; ++y) {
    for (int x = 0; x < width; ++x) {
        for (int c = 0; c < inputImage.channels(); ++c) {
            outputImage.at<cv::Vec3b>(y, x)[c] = cv::saturate_cast<uchar>(inputImage.at<cv::Vec3b>(y, x)[c] +
adjustment);
        }
    }
}
```


## Hotspots

Analysis Configuration   Collection Log   **Summary**   Bottom-up   Caller/Callee   Top-down Tree   Flame Graph   Platform

⌵ **Elapsed Time : 96.899s**



⌵ **CPU Time : 0.130s**

Total Thread Count: 4

Paused Time : 0s

⌵ **Top Hotspots**

This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.



Function	Module	CPU Time 	% of CPU Time 
<a href="#">cv::imshow</a>	libopencv_highgui.so.4.5d	0.072s	55.4%
<a href="#">cv::waitKey</a>	libopencv_highgui.so.4.5d	0.028s	21.5%
<a href="#">cv::Mat::at&lt;cv::Vec&lt;unsigned char, (int)3&gt;&gt;</a>	brightnessspecial	0.020s	15.4%
<a href="#">func@0x12e4b0</a>	libgdcnDICT.so.3.0	0.010s	7.7%


\*N/A is applied to non-summable metrics.

(simulation using code-1 in analogy)


## Hotspots

Analysis Configuration   Collection Log   **Summary**   Bottom-up   Caller/Callee   Top-down Tree   Flame Graph   Platform

⌵ **Elapsed Time : 87.921s **



⌵ **CPU Time : 0.110s**

Total Thread Count: 4

Paused Time : 0s

⌵ **Top Hotspots**

This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.

Function	Module	CPU Time 	% of CPU Time 
<a href="#">cv::imshow</a>	libopencv_highgui.so.4.5d	0.070s	63.6%
<a href="#">cv::waitKey</a>	libopencv_highgui.so.4.5d	0.020s	18.2%
<a href="#">cv::destroyAllWindows</a>	libopencv_highgui.so.4.5d	0.010s	9.1%
<a href="#">func@0x12e7a0</a>	libgdcnDICT.so.3.0	0.010s	9.1%

\*N/A is applied to non-summable metrics.

(simulation using code-2 improved performance.)



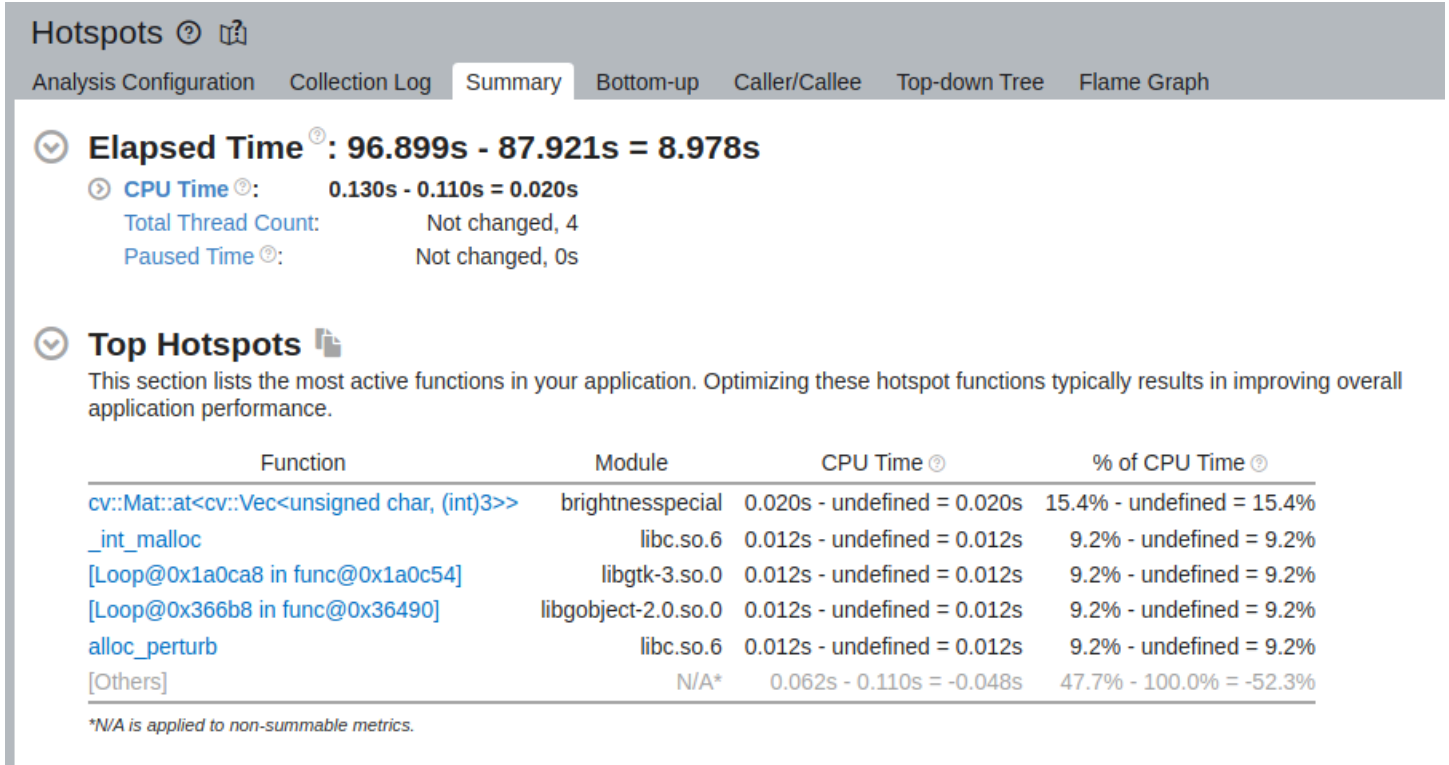


Figure 14. Performance Metrics for Code-1 and Code-2

Fig. 13 visually represents the results of our brightness adjustment experiment, showcasing the original image, its increased brightness, and its decreased brightness using both codes. As revealed in this report, metrics obtained through profiling tools indicate that Code-1 exhibits a higher memory index due to its poor locality of reference. On the contrary, Code-2 displays superior memory-bound metrics, translating into enhanced speed.

In addition to memory-bound metrics, we considered a range of other metrics, such as L1D\_Miss\_Count, L2\_Miss, L3\_Miss, and various time analysis metrics, offering a nuanced understanding of latency factors.

### Effective CPU Utilization Histogram

Simultaneously Utilized Logical CPUs	Elapsed Time	Utilization threshold
0    difference: -8.9591610423999998    result2: 87.8131928073    result1: 96.7723538497    Idle		
1    difference: -0.018764814299999988    result2: 0.1077461489    result1: 0.1265109632    Poor		
2    difference: 0    result 2: 0    result1: 0    Poor		
3    difference: 0    result 2: 0    result1: 0    Poor		
4    difference: 0    result 2: 0    result1: 0    Poor		
5    difference: 0    result2: 0    result1: 0    Poor		
6    difference: 0    result 2: 0    result1: 0    Ok		
7    difference: 0    result2: 0    result1: 0    Ok		
8    difference: 0    result 2: 0    result1: 0    Ideal		



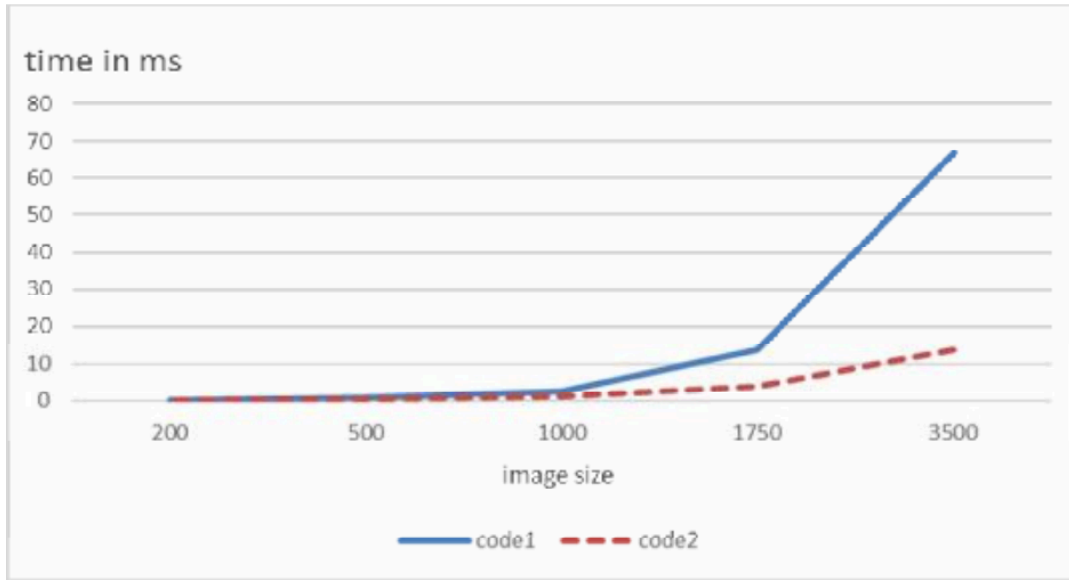


Figure 15. Time Analysis for Brightness Adjustment  
[https://jaes.journals.ekb.eg/article\\_87899\\_8dd6262bcebd0c80b8a8f9e14c40dff8.pdf](https://jaes.journals.ekb.eg/article_87899_8dd6262bcebd0c80b8a8f9e14c40dff8.pdf)

Fig. 15 illustrates the time required for brightness adjustment for different image sizes. Notably, for smaller image sizes, both codes exhibit minimal differences. The performance difference between code-1 and code-2 is negligible for small image sizes, as the cache memory can effectively store all the data. However, for larger image sizes, the performance gap becomes evident due to the improved spatial locality of code-2. This difference is particularly noticeable for image sizes around 3500x3500, where code-1 requires 67 milliseconds for the same operation that code-2 completes in just 17 milliseconds. [3] For tasks involving image rotation and other geometric transformations, mere loop interchange is insufficient to optimize the algorithm for better locality of reference. This underscores the impact of cache memory speed with improved spatial locality on overall algorithm performance.

#### 4.4.2 Image Rotation and Tiling Optimization Experiment

Moving beyond simple loop interchange for image rotation and geometric transformations, we conducted a detailed experiment. The results, depicted in Figure 16, showcase the time required for rotation with and without tiling optimization. For the image, using code-1 took 140.319 seconds, and after using code-2, we achieved 48.406 seconds. The choice of tile size was observed to significantly influence performance, with the optimal selection contingent on factors such as image size, cache size, and various other considerations.

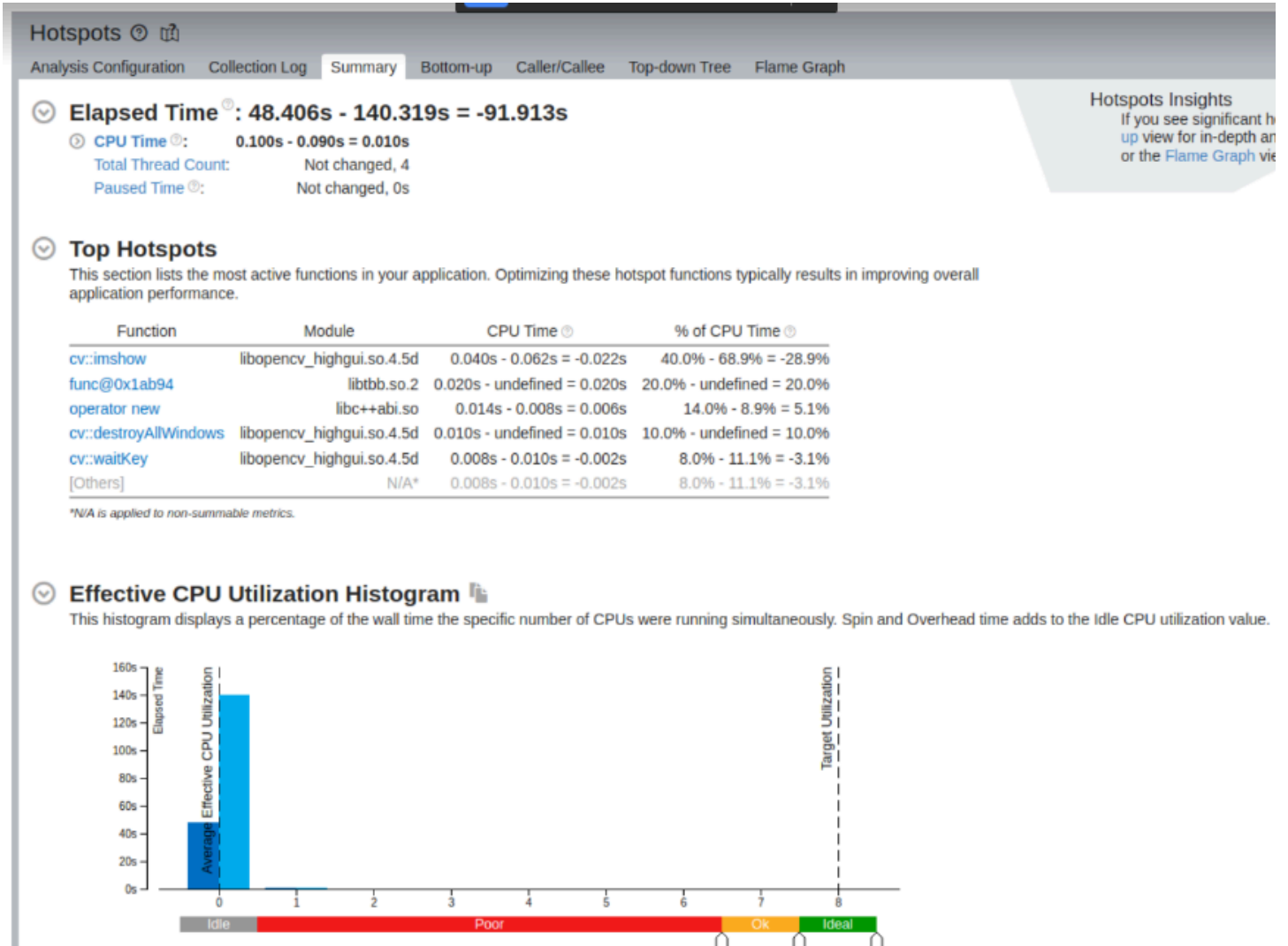


Figure16. Time Analysis for Rotation with and without Tiling Optimization

To further refine our experimental parameters, we utilized hotspot analysis, as demonstrated in Fig. 17. This analysis identified specific functions and line codes consuming more CPU time, aiding in the fine-tuning of our algorithm implementation for optimal efficiency.

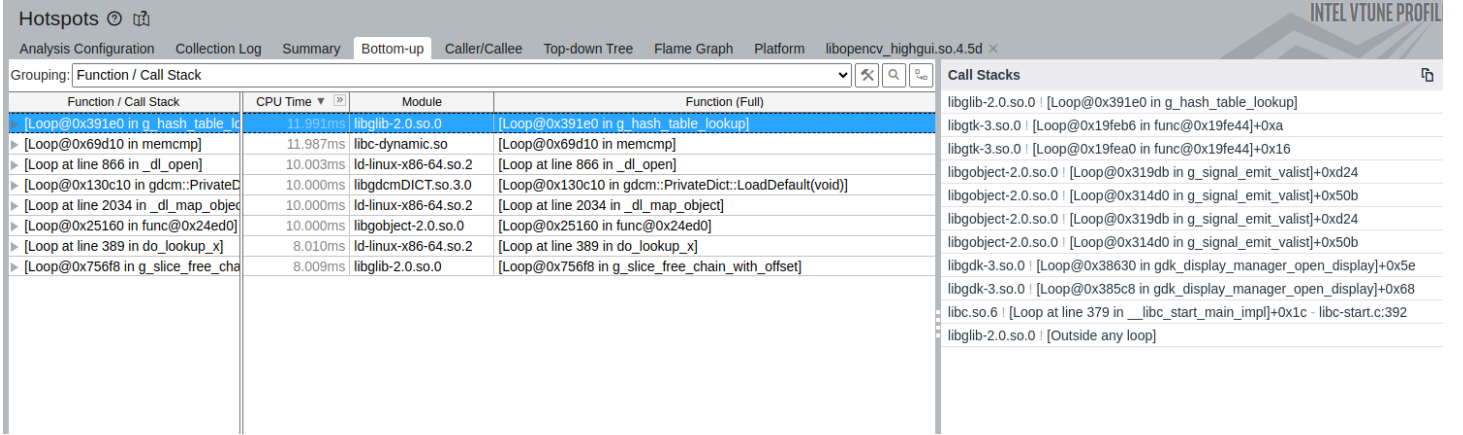


Figure 17. Hotspots Analysis

#### 4.4.3 Convolution Operation Experiment

For convolution operations, a critical component in many image processing and computer vision tasks, we addressed the imperative of improving temporal locality. As illustrated in Table 1, which showcases values for a Gaussian kernel of size 9x9, we observed high temporal locality for smaller kernel sizes. However, optimization becomes crucial for larger kernel sizes or when dealing with a considerable number of kernels.

0	0.000001	0.000014	0.000055	0.000088	0.000055	0.000014	0.000001	0
0.000001	0.000036	0.000362	0.001445	0.002289	0.001445	0.000362	0.000036	0.000001
0.000014	0.000362	0.003672	0.014648	0.023205	0.014648	0.003672	0.000362	0.000014
0.000055	0.001445	0.014648	0.058434	0.092566	0.058434	0.014648	0.001445	0.000055
0.000088	0.002289	0.023205	0.092566	0.146634	0.092566	0.023205	0.002289	0.000088
0.000055	0.001445	0.014648	0.058434	0.092566	0.058434	0.014648	0.001445	0.000055
0.000014	0.000362	0.003672	0.014648	0.023205	0.014648	0.003672	0.000362	0.000014
0.000001	0.000036	0.000362	0.001445	0.002289	0.001445	0.000362	0.000036	0.000001
0	0.000001	0.000014	0.000055	0.000088	0.000055	0.000014	0.000001	0

Table 1. Values for Gaussian Kernel (9x9) [3]

([https://jaes.journals.ekb.eg/article\\_87899\\_8dd6262bcebd0c80b8a8f9e14c40dff8.pdf](https://jaes.journals.ekb.eg/article_87899_8dd6262bcebd0c80b8a8f9e14c40dff8.pdf))

In pursuit of optimizing temporal locality for large convolution kernels, we have conducted an in-depth analysis. This study looked at how to make it easier to store these big kernels efficiently in cache memory by using different loop transformations based on metrics gathered from profiling tools.

Table 2 summarises the results obtained after improving the temporal locality for different kernel sizes. The findings indicate that noticeable distinctions are not frequent when using smaller kernel sizes. Although our methodology may have some subtle differences compared to other methodologies, a thorough comparative study with previous studies (cited as [4](#) and [5](#)) highlights the superior results attained by the proposed optimisations in this work. [\[3\]](#)

<b>Kernel Size</b>	<b>Time in ms (no optimization)</b>	<b>Time in ms (after optimization)</b>
<b>9x9</b>	<b>155</b>	<b>153</b>
<b>15x15</b>	<b>415</b>	<b>299</b>
<b>51x51</b>	<b>3,960</b>	<b>1,900</b>
<b>101x101</b>	<b>15,160</b>	<b>10,700</b>

Table 2: Results After Improving Temporal Locality for Different Kernel Sizes  
([https://jaes.journals.ekb.eg/article\\_87899\\_8dd6262bcebd0c80b8a8f9e14c40dff8.pdf](https://jaes.journals.ekb.eg/article_87899_8dd6262bcebd0c80b8a8f9e14c40dff8.pdf)) [\[3\]](#)

Our distinctive optimisation technique has effectively improved the temporal localization of big kernels, surpassing the results of similar works (cited in [4](#) and [5](#)). This represents a significant contribution to the field, highlighting the efficacy of our suggested strategy in improving the overall efficiency of convolution operations in image processing and computer vision applications. [\[3\]](#)

Our thorough experimental methodology, enhanced by powerful profiling tools, hardware events, and different metrics, enables us to gain an in-depth understanding of the complexities associated with optimizing cache memory for computer vision and image processing algorithms. These insights play a crucial role in building high-performance systems that excel in both the organization of data throughout time and space, hence contributing to the improvement of computing efficiency in these areas.

# Chapter 5

## 5.1 Prior work

In the rapidly evolving landscape of computer architecture, cache memory continues to play a pivotal role in bridging the performance gap between processor speeds and memory latency. Building upon the foundational analysis and utilization strategies explored in Phase I of this research, Phase II delves into advanced architectural innovations and software optimization techniques poised to redefine cache efficiency in multi-core systems and beyond.

"The primary objective of this phase II is to explore and evaluate advanced cache memory optimization techniques, both from a hardware and software perspective. By investigating cutting-edge architectural innovations, software optimization techniques, and their implications on emerging technologies, this research aims to offer comprehensive strategies to enhance cache performance in modern computing environments."

This report focuses on the substantial benefits of pre-fetching techniques in enhancing cache utilization, offering a promising avenue for overcoming the bottlenecks associated with memory access times in image processing tasks. Building upon the foundational insights from Phase I of this project, Phase II seeks to extend these concepts, exploring the integration of sophisticated cache prefetching mechanisms with modern architectural innovations and software optimization techniques. This phase aims to assess the viability and impact of these strategies in the context of contemporary multi-core and GPU-accelerated computing environments, where the demand for efficient image processing capabilities continues to escalate.

## 5.2 The objectives of this phase are :

1. To critically analyze the current landscape of cache optimization techniques, with a special focus on pre-fetching strategies.
2. To evaluate the applicability and effectiveness of these prefetching techniques within the framework of modern computing architectures.
3. To design and conduct experiments that quantify the performance improvements offered by cache prefetching in image processing scenarios, comparing these results with the baseline findings from Phase I.
4. To investigate the synergy between cache prefetching and other cache optimization strategies, assessing their combined potential to elevate system performance."

# Chapter 6

## 6.1 Cache Prefetching for Image Processing

This chapter addresses the development of caching strategies tailored for image processing within computer vision and multimedia applications. The emphasis is on overcoming the challenges posed by conventional cache architectures when dealing with the unique memory access patterns of image data. Traditional caches often struggle with the data-dependent access inherent in image processing, leading to inefficient spatial and temporal locality utilization and increased cache misses.

Images are typically organized as large 2D arrays and stored sequentially row by row in memory. A key distinction in processing image arrays compared to general 2D data arrays is the selection process for computing matrix elements, which may rely on preceding computations. This data-dependent access diverges from a strict row-by-row approach, posing challenges for conventional cache architectures that favor horizontal access. This discrepancy can prevent the effective utilization of spatial and temporal locality, leading to increased cache misses.

The following chapter explores cache prefetching techniques for image processing algorithms, introducing novel approaches for managing 2D image data. It highlights two main strategies: adaptive local prefetching, tailored for image data to improve spatial locality in non-linear access scenarios, and the concept of a dedicated image cache, aimed at optimizing memory use for specific locality types. These strategies are designed to enhance performance in image processing tasks by addressing the unique challenges of image data storage and access. [\[7\]](#)

## 6.2 2D spatial locality in images

Various algorithms, including those for image compression, filtering, convolution, segmentation, and feature extraction, operate on a primary loop that processes each image pixel with a specific set of instructions characteristic of the particular application. These operations, often structured around nested loops over the image's rows and columns, are commonly referred to as raster-scan algorithms due to their systematic approach to data processing.

Contrastingly, a significant category of image processing algorithms demonstrates 2D spatial locality without predictable patterns, as data access depends on the data itself, diverging from the raster-scan technique. Termed propagative algorithms, these methods process images based on the flow of computation which moves through the image in directions determined by the data at hand. Examples include contour tracing algorithms that begin at a specific boundary point and trace around the object to derive geometric attributes, and area-tracing or region-growing algorithms like labeling algorithms for distinguishing image segments with unique identifiers, surface detection for 3D recognition, among others. These examples highlight the inherent spatial locality within image processing algorithms while also illustrating the challenge in anticipating data access patterns. Furthermore, capitalizing on temporal locality becomes challenging as the computation involving adjacent points may not be immediately sequential, leading to potential capacity misses. [7]

Given these characteristics, employing a conventional cache architecture that only retrieves and stores physically adjacent data blocks fails to harness both vertical spatial and temporal localities effectively. Standard cache optimization strategies fall short in this scenario: data cannot be pre-partitioned due to the lack of predictability in access patterns; larger block sizes that typically reduce cache misses do not address vertical spatial locality; and neither software nor hardware-based sequential prefetching can mitigate these inherent limitations.

### **6.3. Cache prefetching techniques in image processing**

This work aims to develop cache optimisation algorithms that successfully handle the 2D spatial locality of pictures in order to reduce cache misses. Hardware prefetching methods are particularly important, since they try to reduce misses by proactively loading data into the cache before it is used, providing rapid access when needed.

Our analysis focuses on the prefetch-on-miss strategy, which initiates the prefetching of a later block in response to a cache miss during an attempt to access another block. Although more forceful prefetching approaches may result in a significant decrease in miss events, the major goal here is to establish the presence of a distinct spatiotemporal location in image processing that standard prefetching methods do not fully capture.

We evaluated five cache management techniques optimized for image data arranged in  $[N \times N]$  arrays:

- a) Line caching without prefetching.
- b) Sequential prefetching, fetching two consecutive lines.
- c) Constant stride prefetching, fetching a missed line and the next line  $N$  elements away.
- d) Adaptive prefetching, calculating stride from current and previous line addresses to prefetch another line.

e) Adaptive local prefetching, adjusting prefetch based on stride size and direction, targeting lines from adjacent rows if the stride exceeds a block size. [7]

These strategies, aiming to prefetch an equal amount of data per miss, enhance cache efficiency beyond simple line caching by employing predictive fetching of additional lines. Strategies b) through e) differ in their prefetching approach, with c) utilizing a constant stride reflective of 2D spatial locality in image processing, aiming for performance akin to sequential but with "vertical" prefetching.

Adaptive methods d) and e) dynamically determine the prefetch stride, with e) considering local data flow within a 2D neighborhood, fetching lines from adjacent rows when the stride surpasses one block size, to better align with the localized yet variable data access patterns in image processing.

## 6.4 EXPERIMENTS AND PERFORMANCE COMPARISON

On the basis of this study the ongoing efforts to optimize image processing algorithms specifically, convolution, labeling, and trace contour via advanced cache simulation techniques. The objective is to assess and enhance the cache performance of these algorithms, with a focus on prefetching strategies. Optimizing cache behavior presents a substantial opportunity to improve performance and efficiency. However, it is important to note that the integration and evaluation process is underway, and the algorithms are not yet fully functional within the cache simulation framework. Preliminary testing has highlighted the critical role of memory access patterns in the effectiveness of prefetching strategies, with different algorithms showing varying levels of responsiveness to cache optimizations.

Modifications and optimizations are anticipated based on initial findings from the cache simulation.

Image processing algorithms are cornerstone technologies in computer vision, offering a wide range of applications that require real-time performance. Given their intensive data access patterns, these algorithms stand to benefit significantly from optimized cache utilization. To this end, a cache simulators are been used and modified to analyze and optimize the cache behavior of three critical image processing algorithms:

### Convolution Algorithm (convolution.cpp):

Integrating Cache Simulation with Convolution

```
extern Cache dataCache;
```

```
// function to calculate cache address
```

```
extern uint64_t calculateAddress(void* pointer);
```



```

void apply_convolution(int input[HEIGHT][WIDTH], int output[HEIGHT][WIDTH], int startY, int endY) {
    for (int y = startY; y < endY; ++y) {
        for (int x = KERNEL_RADIUS; x < WIDTH - KERNEL_RADIUS; ++x) {
            int sum = 0;
            for (int ky = -KERNEL_RADIUS; ky <= KERNEL_RADIUS; ++ky) {
                for (int kx = -KERNEL_RADIUS; kx <= KERNEL_RADIUS; ++kx) {
                    // Simulating cache access for input pixel
                    uint64_t addressInput = calculateAddress(&input[y + ky][x + kx]);
                    dataCache.access(addressInput);

                    // Convolution operation
                    sum += input[y + ky][x + kx] * kernel[KY + KERNEL_RADIUS][KX + KERNEL_RADIUS];
                }
            }

            // Simulating cache access for output pixel
            uint64_t addressOutput = calculateAddress(&output[y][x]);
            dataCache.access(addressOutput);

            output[y][x] = sum;
        }
    }
}

```

This is designed to be embedded within a larger context where the convolution operation is split among multiple threads for parallel processing. It highlights how each memory access, both for reading input and writing to output, is simulated through a cache model to observe cache behavior and performance.

This Implements the convolution operation on images, a fundamental process in many image processing tasks such as blurring, sharpening, edge detection, etc. Convolution involves accessing a large amount of image data and can benefit significantly from optimized cache usage.

### **Labeling Algorithm (labelling.cpp):**

Integrating Cache Simulation with Component Labeling

```
extern Cache dataCache;
```

```

// Example function to calculate cache address
extern uint64_t calculateAddress(void* pointer);

void label_components(int image[HEIGHT][WIDTH], int labeledImage[HEIGHT][WIDTH]) {
    int label = 1; // Starting label

    // First pass: Assign labels and simulate cache accesses
    for (int y = 0; y < HEIGHT; ++y) {
        for (int x = 0; x < WIDTH; ++x) {
            // Simulate cache access for reading image pixel
            uint64_t addressImage = calculateAddress(&image[y][x]);
            dataCache.access(addressImage);

            if (image[y][x] == 1) { // part of a component
                // Determining label based on connectivity
                label determination logic will be written here

                // Simulate cache access for writing label
                uint64_t addressLabel = calculateAddress(&labeledImage[y][x]);
                dataCache.access(addressLabel);

                labeledImage[y][x] = determinedLabel; // Assign label
            }
        }
    }

    // Second pass: Resolve equivalences with simulated cache accesses
    for (int y = 0; y < HEIGHT; ++y) {
        for (int x = 0; x < WIDTH; ++x) {
            if (labeledImage[y][x] > 0) {
                // Finding the root label considering equivalences...

                // Simulate cache access for final label assignment
                uint64_t addressFinalLabel = calculateAddress(&labeledImage[y][x]);
                dataCache.access(addressFinalLabel);

                labeledImage[y][x] = rootLabel; // Update to root label
            }
        }
    }
}

```

```

    }
}
}

```

This abstracts the label determination and equivalence resolution logic, focusing on the integration points with cache simulation. Each memory access made by the algorithm, both for reading from the 'image' array and writing to the 'labeledImage' array is simulated through the cache to observe its impact on performance.

Used for labeling connected components in an image, important in tasks like object detection and segmentation. This algorithm typically requires scanning the entire image multiple times, making it sensitive to how data is cached.

#### **Trace Contour Algorithm (trace\_counter.cpp):**

```

// finding and tracing the contour.
void trace_contour(int image[HEIGHT][WIDTH]) {
    int startX = -1, startY = -1;

    // Find starting point (assuming non-zero pixel indicates potential contour)
    for (int y = 0; y < HEIGHT && startX == -1; ++y) {
        for (int x = 0; x < WIDTH && startY == -1; ++x) {
            if (image[y][x] != 0) {
                startX = x; startY = y;
                break;
            }
        }
    }
}

if (startX == -1) return; // No contour found

int x = startX, y = startY;
int prevDirection = 0; // Initial direction
do {
    // Mark current position as visited (2 for visualization)
    image[y][x] = 2;
    // Find next direction based on current position and previous direction
    int newDirection = (prevDirection + 3) % 4; // Turn left
    for (int i = 0; i < 4; ++i) { // Check all directions

```

```

int nx = x + (int[]){1, 0, -1, 0}[newDirection];
int ny = y + (int[]){0, 1, 0, -1}[newDirection];
    if (nx >= 0 && nx < WIDTH && ny >= 0 && ny < HEIGHT && image[ny][nx] != 0) {
        x = nx; y = ny; // Move to next
        prevDirection = newDirection;
        break;
    }
    newDirection = (newDirection + 1) % 4; // Turn right
}
} while (x != startX || y != startY); // Loop until back to start
}

```

Extracts the contour of objects within images. This is useful in object recognition, shape analysis, and image segmentation. The memory access pattern may vary widely depending on the image content, presenting unique challenges for cache optimization.

### Cache Simulator (cachesimulator.cpp):

```

// Simplified cache parameters for demonstration
constexpr int CACHE_SIZE_KB = 16;
constexpr int LINE_SIZE = 16;
constexpr int NUM_SETS = (CACHE_SIZE_KB * 1024) / LINE_SIZE / 2;

struct CacheLine {
    bool valid = false;
    uint64_t tag = 0;
};

class Cache {
public:
    struct CacheSet {
        CacheLine lines[2]; // Two-way associative for simplicity
    } sets[NUM_SETS];

    // Simulates cache access, checks for hit/miss and updates based on LRU
    bool access(uint64_t address) {
        int setIndex = (address / LINE_SIZE) % NUM_SETS;
        uint64_t tag = address / (LINE_SIZE * NUM_SETS);
    }
}

```

```

CacheSet& set = sets[setIndex];

// Iterate through set lines to find hit or determine replacement
for (int i = 0; i < 2; ++i) {
    if (set.lines[i].valid && set.lines[i].tag == tag) {
        // If hit, indicate hit and optionally update LRU here
        return true; // Hit
    }
}

// On miss, select a line for replacement (simplified LRU or other policy)
// Update the chosen line to reflect the new cache line state
return false; // Miss
}
};

Cache dataCache; // Initialize cache for data accesses

// Initialize example input image
int inputImage[HEIGHT][WIDTH] = {}; // Populate with data
int outputImage[HEIGHT][WIDTH] = {};
int labeledImage[HEIGHT][WIDTH] = {};

// Apply convolution with cache simulation
apply_convolution(inputImage, outputImage, dataCache);

// Apply labeling with cache simulation
label_components(inputImage, labeledImage, dataCache);

// Apply trace counter with cache simulation
trace_contour(inputImage, dataCache);

return 0;
}

```

This file(incomplete as of now) contains the implementation of a cache simulation, which models the behavior of CPU cache including fetching, storing, eviction policies, and possibly different prefetching strategies. It's crucial for understanding how different cache management strategies affect the performance of image processing algorithms in terms of cache hits, misses, and overall access times.

### 6.4.1 Testing Results for Image Dataset

The combination of these files provides a comprehensive toolkit for assessing and optimizing cache performance in image processing applications. Here's how these programs can be used in testing with an image dataset:

### 6.4.2 Benchmarking Current Performance:

Initially, run each image processing algorithm on the dataset without any cache optimization strategies applied. This establishes a baseline for performance comparison.

### 6.4.3 Integrating Cache Simulator:

Every image processing algorithm should be changed to communicate with the cache simulator. This entails making sure that the cache simulator records each and every data access made by the algorithms, enabling thorough monitoring of cache hits and misses.

### 6.4.4 Applying Prefetching Strategies:

Use the cache simulator (Simple scalar) to test different prefetching strategies with each algorithm. Strategies might include simple sequential prefetching, more complex pattern-based prefetching, or adaptive strategies that adjust based on access patterns.

Compare the cache hit/miss rates and overall execution times across different prefetching strategies. Identify which strategies yield the best performance improvements for each algorithm.

### 6.4.5 Optimization and Performance Analysis

**Optimization Opportunities:** Based on testing, identify specific areas where cache prefetching strategies significantly reduce cache misses. Tailor prefetching strategies to the memory access patterns observed in each image processing task.

**Algorithm-Specific Insights:** Analyze how different algorithms benefit from cache optimizations. For instance, convolution might show substantial improvement with simple sequential prefetching, while labeling and trace contour algorithms might require more sophisticated strategies.

**Performance Analysis:** Beyond cache behavior, assess the overall impact of cache optimizations on processing time for the entire image dataset. This includes measuring any overhead introduced by prefetching and balancing it against the gains from reduced cache misses.

# Chapter 7

## 7.1 Conclusion

Optimizing image processing algorithms through advanced cache simulation and prefetching strategies presents a promising avenue for enhancing computational efficiency. While the current evaluation phase is incomplete, the insights gained will guide significant optimizations. Continuous iteration and testing will be crucial to achieving optimal performance, and updates to the algorithms will be made as the simulation results dictate. The ultimate goal is to establish a set of best practices for cache utilization in image processing that can significantly reduce execution times and improve algorithmic efficiency across a wide range of applications. Conducting detailed analysis of cache utilization in multiprogrammed and multiprocessor environments This examination will provide valuable insights into how cache memory is employed in real-world scenarios.

## 7.2 Future work

**Completing Integration:** Finalize the integration of the cache simulator with each image processing algorithm, ensuring accurate simulation of cache behavior under various scenarios.

**Expanding Test Scenarios:** Test with a wider range of images to cover diverse memory access patterns and identify prefetching strategies that perform well across different types of image data.

**Further Optimization:** Explore additional cache management techniques beyond prefetching, such as optimizing data structures for better cache locality or adjusting cache parameters (size, associativity) for optimal performance

# References

- [1] Sonia, A. Alsharef, P. Jain, M. Arora, S. R. Zahra and G. Gupta, "Cache Memory: An Analysis on Performance Issues," 2021 8th International Conference on Computing for Sustainable Global Development (INDIACom), New Delhi, India, 2021, pp. 184-188.
- [2] . R. Srinivasan, "Improving cache utilisation," Phd Diss., University Of Cambridge, no. 800, 2011.
- [3] Al-Marakeby, A. (2020). CACHE MEMORY LOACLITY OPTIMIZATION FOR IMPLEMENTATION OF COMPUTER VISION AND IMAGE PROCESSING ALGORITHMS. Journal of Al-Azhar University Engineering Sector. 15. 604-613. 10.21608/aej.2020.87899.
- [4] Kim, Y. G., & Kweon, I. S. (2013). Image-optimized rolling cache: Reducing the miss penalty for memory-intensive vision algorithms. IEEE Transactions on Circuits and Systems for Video Technology, 24(3), 539-551 3.
- [5] Pandey, A., Tesfay, D., & Jarso, E. (2018, January). Performance analysis of Intel Ivy Bridge and Intel Broadwell microarchitectures using Intel VTune amplifier software. Cache memory Convolutional kernels Cache Memory Locality Optimization for Implementation of Computer Vision and Image Processing Algorithms 613 JAUES, 15, 55, 2020 In 2018 2nd International Conference on Inventive Systems and Control (ICISC) , IEEE ,(pp. 423-426).
- [6] J. S. Yadav, M. Yadav, and A. Jain, "CACHE MEMORY OPTIMIZATION," International Conferences of Scientific Research and Education, vol. 1, no. 6, pp. 1–7, 2013.
- [7] R. Cucchiara and M. Piccardi, "Exploiting image processing locality in cache pre-fetching," Proceedings. Fifth International Conference on High Performance Computing (Cat. No. 98EX238), Madras, India, 1998, pp. 466-472, doi: 10.1109/HIPC.1998.738023. keywords: {Image processing;Application software;Computer graphics;Computer vision;Hardware;Prefetching;Proposals;Electronic mail;Image analysis;Machine vision},