CHAPTER 4

PROPOSED CONTENT PROXY SERVER
WITH IAA TREE INDEXING

The multimedia content adaptation process has been enhanced with the implementation of a proxy server, which reduces the load on the server and executes possible operations in the intermediate level itself. The proxy server has been designed to serve the request from the client, which figures out the possibility of using the existing relevant content available in the cache. The time required for retrieval of relevant information has been minimized by maintaining the cache in the proxy server which could hold a considerable amount of recently accessed information objects. Such cache must not confine to only one of the multimodal objects, but rather provide an overall caching of multimedia objects which can effectively serve the request from the client.

Proposed content proxy server with cache reduces the load on the server in terms of processing the request as well as storage and retrieval of objects. The response time has been reduced by serving the content from an intermediate server which eventually reduces the network transfer and the retrieval time of the frequently requested information. The efficiency of the proxy server has been increased manifold with the implementation of the proposed indexing mechanism for better storage and retrieval of information. Proposed Image Adaptation Aware Indexing (IAA) Tree mechanism has been designed to consider the adaptation coefficients of the objects and store relative objects closer to enable faster retrieval.
4.1 CONTENT PROXY ARCHITECTURE

The content adaptation process can be performed in various methods to improve the efficiency of multimedia content retrieval suitable to the user device characteristics. The content adaptation architecture for pervasive systems (Berhe et al., 2005) has been shown in Figure 4.1. The architecture consists of local proxies to serve the request from the client devices, which interact with the content server to retrieve the information. Adaptation Services and Adaptation Service Registry present in the system along with client profile database perform the required content adaptation process on the multimedia information.

Figure 4.1 Distributed Content Adaptations for Pervasive Systems (Berhe 2005)
The functions of various components present in the system are as follows

(i) Local Proxies

They intercept the user request and server responses along with initiating the transfer of adapted content.

(ii) Content Proxies

They accept user requests forwarded by the local proxies and retrieve the images from either the local cache or remote content servers. Adaptation has been performed in the adaptation engine if the required image version is unavailable in the local cache and then transferred back to the local proxies.

(iii) Adaptation Service Proxy

Adaptation Service Proxies (ASP) are the web services that can be deployed on the content proxies to execute the required adaptation operations. They carry over the necessary operation on the multimedia content and deliver the content in a suitable format as per the characteristics of the user device.

(iv) Profiles

The architecture supports Device profiles for storing device capability and compatibility information, Client profiles for storing the user preferences and display settings. The Adaptation Service Registry stores the service profiles and provides required instructions to the ASP.

Implementing cache at any point in this architecture increases the productivity of the system. Though cache makes sense both at local and content proxies, the maximum performance gain can be attained when the cache has been used at the independent content proxies, as it provides all the aforementioned gains. A second driving force behind this decision is that the local proxies are constrained in the current scenario of mobile computing.
4.2 WORKING OF CONTENT PROXY

The Query Processor (QP) and Adaptation Engine (AE) perform the decision making with respect to adaptation operations that need to be performed on appropriate multimedia objects. On the other hand, the Cache Management module (CM) maintains cache index and manages replacement operations based on the input parameters obtained from the above mentioned two modules. The overall working of the proxy can be described as follows.

i. Communication Interface gets the client request containing identifiers for the client, requested content and the device type.

ii. QP deserializes the query and forwards the parameters to CM.

iii. CM tries to fetch the transcodable version of the requested object in the cache and returns the content, if available.

iv. If the content is unavailable in the cache, then the request is forwarded to AE.

v. AE fetches the device profile based on device type specified in the query and decides on the adaptation operations.

vi. After the required adaptation process, the adapted content is forwarded to QP.

vii. QP forwards the resultant content to the CM.

viii. CM updates the content version with the required parameters into the cache and accordingly updates the indexing structure present in the cache.

ix. Cache overflow has been avoided by removing the least preferable image using the replacement policy.
4.3 ADAPTATION AWARE CACHING SCHEME

Cache efficiency is characterized by the amount of less important cached images, size of allotted memory and the time required for transmission of data from the remote server to the content proxy. As the indexing scheme being implemented in the proposed system focuses on being adaptation aware, the first two factors can be satisfied by implementing a suitable indexing mechanism. For content adaptation proxies, similar objects should be indexed closer. Moreover, the index should have caching policy information in order to achieve faster image replacement in case of a cache miss. And as per convention, the removal policy in the index should favour more frequently accessed objects which can lead to a reduction in computational overheads. Consequently, the response time can be improved.

A cache management algorithm can be analogized to the classic Knapsack problem (Kitamoto and Asanobu 2000). Consider \( D \) as the dataset (more specifically image set in this case) given in Equation (4.1).

\[
D = \{D_1, D_2, \ldots, D_N\}
\] (4.1)

\[
x_i = \{0,1\} \text{ where } x_i = 0, \text{ if } D_i \text{ is cached}
\]

\[
x_i = 1, \text{ if } D_i \text{ is not cached}
\]

The aim of the caching mechanism has been described in Equation (4.2) and (4.3).

a. Maximize \( V = \Sigma_{i=1}^{N} x_i w_i \) (4.2)

b. Ensure \( \Sigma_{i=1}^{N} x_i s_i \leq S \) (4.3)

where \( S \) is the total cache size, \( V \) is the tree vertices, \( w_i \) and \( s_i \) are said to be the “relevancy value” and size of \( D_i \) respectively.
It has been explained from the above expressions that the “relevancy value” \( w \) plays a pivotal role in the overall design and implementation of the cache. Keeping in mind the application of caching on content adaptation proxies, the relevancy value for a particular version of the image in the cache can be defined as Equation (4.4)

\[
w_{ij} = F(s_{ij}, d_{ij}, f_{ij}, t_{ij}, n_i)
\]  

(4.4)

where,

- \( s_{ij} \) is the size of image version
- \( r_{ij} \) is the image version resolution
- \( d_{ij} \) is distance of the image version from “nearest similar” image
- \( f_{ij} \) is frequency of access of image version
- \( t_{ij} \) is the last access time of image version
- \( n_i \) is the number of cached image versions of \( D_i \)

From the above notations, it has been observed that a crucial parameter in determining the relevancy value in content proxy cache is the distance of the image version from the “nearest similar” image. The nearest similar image can be defined as another version of the same image that has higher resolution and from which the target image can be obtained through local adaptation. Arriving at a suitable value for this parameter can result in the reduction of repeated requests to the remote server for the same image version. Some of the other terms that play an important role to describe the caching mechanism has been defined below:

- **a. Cache Hit**: When an exact match for the user request has been found in the cache, it is termed as a cache hit. This depends on the availability of frequently accessed cache images in the cache, which has been directly influenced by the replacement policy.
b. Cache Miss: When the requested image is not found in the cache, then it is termed as Cache Miss. In the event of unavailability of the required content in the cache, it has to be retrieved from the content server. This involves IO time, hence there is an increase in the response time.

c. Partial cache hit: Partial cache hit can be defined as a scenario in which the requested image is not found, but an image which could be an acceptable replacement, or the one from which the object could be obtained by adaptation is present. Here the content server need not be accessed and hence the retrieval time is comparatively reduced.

d. Threshold: The threshold plays a major role when a partial cache hit occurs, to ascertain the limits of acceptability of cache images being used for providing the requested object. In addition, the parameter determines when an object in the cache can be used for serving the user request with slight adaptation, thereby eliminating the need to contact content server. The threshold value is not maintained as a constant, as this value depends on the processing capability of the deployment system, the promised QoS and other such factors that depends on individual instances. Hence it is maintained as a variable parameter.

4.4 CACHE ACCESS SCENARIOS

There are three possible scenarios that can occur when a client request for an image object that arrives at the content proxy – Cache Hit, Partial Cache Hit and Cache Miss.
4.4.1 Cache Hit

Cache hit scenario occurs when the requested image is present in the cache. In this case, the image has been returned as such and does not involve any adaptation. When a request has been received, the cache management module triggers the operation for retrieving the requested content. The object with same attributes found in the cache is returned to the adaptation engine which is then delivered to the user. The interaction between the various modules involved is shown in Figure 4.2.

Figure 4.2 Cache Hit

The request from the client arrives at the Query processor (see Figure 4.2). The Query processor deserialises the request and pass parameters to the indexing component of cache management. The indexing component of the cache management system searches for the pertinent object. Once the object has been found, the result gets transferred to the query processor and then to the communication interface, from where it reaches the user.
4.4.2 Partial Cache Hit

When the content requested is unavailable in the cache, but another variation of the same image (variations in certain other parameters such as resolution or format) is present in the cache, a partial hit has said to be occurred.

If the cache resident object falls under the transcodable threshold of being adapted to obtain the desired output, then the corresponding object has been retrieved. It is then transcodable to the required format and sent to the user. Thus without the help of content server, the request has been dealt entirely through the proxy. The exchanges taking place in this process have been sketched in Figure 4.3.

![Figure 4.3 Partial Cache Hit](image-url)
The request from the client arrives at the Query processor. The Query processor deserialises the request and passes the required parameters to the indexing component of cache management. The indexing component of the cache management system searches for the requested object. If the target image is not found, then other related version from which the desired object could be transcoded is looked up in the cache. The existence of such an image is decided based upon a threshold value. The candidate object is then sent to the adaptation engine along with the deserialised parameters. The extracted object is adapted according to the parameters and then transferred to the communication module in order to forward it to the user device.

4.4.3 Cache Miss

If neither the target image nor any transcodable variations of it are present in the cache, then there is said to be a cache miss and the server has to be accessed to retrieve the target image. The detail of the scenario is given in the Figure 4.4.

The request from the client arrives at the Query processor. The Query processor deserialises the request and passes the information to the indexing component of the cache management. The indexing component of the cache management system searches for the pertaining image. If no content that falls within the threshold limit has been found in the cache, then the content server is requested for the corresponding image. If the server responds with an exact match, it gets transferred to the user through the communication module.
In case the responded image is not an exact match, the received object is forwarded to the adaptation engine for transformation according to the user request and then sent to the communication interface. In the latter case, the cache must again be updated with the new adapted image. This is based on whether an image or its variation is more frequently accessed.

4.5 IAA TREE DESIGN

The proposed Image Adaptation Aware tree (IAA tree) has been built on the framework of the B+ tree and its cache conscious variants. These trees have been the area of focus for contemporary indexing mechanisms since they provide faster access time and fulfill the efficiency requirements needed for any caching scheme.
The indexing tree scheme has non-leaf nodes, leaf nodes and Image Version Graphs (IVGs) as its major components. Each non-leaf node has multiple references to various child nodes which are subjected to a predefined limit that has been determined by the cache block size. Each leaf node in turn has references to IVGs. Image Version Graphs provide grouping feature for versions of an image and assist in version control within the cache resident image versions. Figure 4.5 shows a sample IAA tree layout with depth = 1.

![IAA Tree Layout Diagram]

**Figure 4.5 IAA Tree Layout**

IAA tree consists of two major components: the base IAA tree and the IVG graph. The IAA tree nodes have been classified as leaf and non-leaf nodes. Each leaf node consists of references to either other non-leaf nodes or leaf nodes. The number of references has been limited by the cache block size, to facilitate maximum performance by optimizing disk access. In addition to the keys and references, the nodes hold the least cost value of their child nodes. The least cost value is the cost of the entry in the child node which represents the next candidate for replacement. The leaf nodes store references to the IVG graphs and the corresponding least cost values.
The IVG graphs are optimal least spanning trees that reflect the transcoding relationships between the variants of an image in an Euclidean space metric representation. This proposed feature, taking into account the transcodability of each image helps the underlying data structure to perform better than the conventional models. This distance metric has been used while choosing the candidates for cache replacement, wherein the nodes with least cost can be removed without affecting the system performance to a greater extent. They can be reproduced by relatively less intensive computation processes during partial cache hits.

Each IVG node consists of a reference to the image inside the cache, the frequency of access of the image, the timestamp corresponding to the latest access of the image and the number of times that the image has served as a parent, i.e. the number of times the image has been transcoded into the required image during partial cache hits. Each edge has been assigned a weight that indicates the transcoding costs to obtain the image from its nearest image among the other members of the IVG graph.

The least cost of the nodes in the tree has been updated during insertion or removal of images. When an image has been inserted into an IVG, or removed from an IVG to make space for a new image, the least cost of the associated IVG changes in case the removed or inserted image has been the image of the least cost (in the case of removal, it is always so).

The IVG root node collects the least cost among the other nodes of the IVG graph and this is sent to the leaf node pointing to it. The leaf node in turn finds the least cost among the costs from its ‘child’ IVGs and stores it as its least cost. The non leaf nodes do the same process for their children. The least cost propagates from the IVG to the root node of IAA in this fashion, making the least cost among all the images in the cache readily available. This
provides the tree with a bimodal search feature. The “least cost mode” can be utilized while handling cache overflow conditions.

**Figure 4.6 IAA tree structure**

The detailed node structure and fields of IAA tree have been depicted in Figure 4.6. The leaf node and non-leaf node can be explained in terms of object references and least cost value given as Equation (4.5) to (5.1).

\[
L_{\text{iaa}} = (\{I_{\text{name}}, LC_{\text{avg}}, \text{Ref}_{\text{avg}}\}, LC_{\text{global}}) \tag{4.5}
\]

\[
NL_{\text{iaa}} = (\{I_{\text{name}}, LC_{\text{avg}}, \text{Ref}_{\text{iaa-node}}\}, LC_{\text{global}}) \tag{4.6}
\]

where,

\[
I_{\text{name}} \quad \text{– Image name}
\]

\[
LC_{\text{avg}} \quad \text{– Least cost of transcoding for } I_{\text{name}}
\]

\[
\text{Ref}_{\text{avg}} \quad \text{– Reference to Image Version Graph}
\]
$\text{LC}_{\text{global}}$ – Least cost in the IAA-node

Image Version Graph has been defined as –

$$G_{ivg} = (V_{ivg}, E_{ivg}, LC_{ivg})$$  \hspace{1cm} (4.7)

where,

$V_{ivg}, E_{ivg}$ are IVG’s vertex list and edge list respectively.

$V_{ivg} = \{C_e\}$, set of cache entries for cache resident image objects

$C_e = (I_{id}, f, t, n_{\text{trans}})$  \hspace{1cm} (4.8)

where,

$I_{id}$ – Image object identifier

$f$ – Frequency of access

$t$ – Last access timestamp

$n_{\text{trans}}$ – Number of local transcoding operations performed on the image object

$E_{ivg} = (v_i, v_j, c_{ij})$  \hspace{1cm} (4.9)

where, $v_i, v_j$ are IVG nodes joined by the edge

$$c_{ij} = \text{res}(v_i) - \text{res}(v_j)$$  \hspace{1cm} (4.10)

In other words, $c_{ij}$ is the edge cost defined as the transcoding cost between $v_i$ and $v_j$.

$$LC_{ivg} = \min \left( \text{cost}(E) \right), \text{ for all } E \in \{E_{ivg}\}$$  \hspace{1cm} (4.11)

This gives the least cost corresponding to the Image Version Graph.
The version frequency of access $f$, last access timestamp $t$, number of local transcoding operations $n_{trans}$ and the IVG edge cost $c_{i,j}$ are the parts of the index parameters. They have been used to determine the appropriate image version to be removed from the cache in the event of a cache miss. In this way, the cache indexing has been taken care of by the IAA tree and the corresponding cache replacement policy has been implemented in consonance.

4.6 IAA TREE ALGORITHMS

A set of algorithms have been proposed for the operation of IAA tree. This includes searching an object by its name when going through the cache, searching a multimedia object using its least cost value to find a related transcodable version of the object, inserting an object into the cache and deleting objects from the cache using cache replacement policy during buffer overflow till space for the new incoming object has been available.

4.6.1 IAA search by name

The first algorithm IAA search by name performs a traversal of the IAA tree to find the IAA node corresponding to the searched image object and then routed down to the appropriate version in the IVG as per the supplied image id. The algorithm IAA search by name has been used for performing a lookup based on the image name that constitutes the image id passed as input. The search begins at the root node of the input IAA tree.

In case the tree is empty, the above algorithm indicates that the cache is empty, by returning null. If there are images resident in the cache, and the node currently under consideration is not a leaf node, the algorithm searches for entries in the current node t to determine the appropriate child node to be traversed, using binary search(). If the current node is a leaf node
and there are no IVG references corresponding to the image being searched, then a cache miss occurs. If not, the algorithm finds the image version “nearest” to the image specified in the request.

**Algorithm: IAA search by name**

input: img-id, tree

output: cache-entry or null

```
Function IAA search by name ()
if(tree[root] = null)
then return null //cache-empty
end if

curr-node := tree[root]
while(curr-node != leaf-node)
curr-node := binary-search(curr-node, img-id[name])
//returns pointer to node corresponding to name
end loop

ivg-ref := binary-search(curr-node, img-id[name])
if(ivg-ref = null) //cache-miss
then return null
end if

cache-entry := find-nearest(img-id, ivg-ref)
if(cache-entry = null) //cache-miss
then return null
else return cache-entry
end if
```

End
4.6.2 IAA search by cost

In order to provide bimodal search feature, the second algorithm IAA search by cost has been proposed for implementing the cache replacement policy in the event of a cache overflow. The Least Cost (LC) global value at the root node denotes the cost corresponding to the candidate object for removal. As in the prior algorithm, the condition of the root being null indicates an empty cache. Each node is traversed until the leaf node is reached. In each iteration, the node corresponding to LC global has been selected for the next iteration. This continues till the IVG that contains the candidate object has been reached. The candidate object has been then extracted from the IVG by comparing the edge costs with LC global.

Algorithm: IAA search by cost

input: tree
output: cache-entry or null

Function IAA search by cost ()

if(tree[root] = null)
then return null //cache-empty
end if

curr-node := tree[root]
while(curr-node!=leaf-node)
curr-node := search-cost(curr-node, curr-node[lc-global])
end loop
ivg-ref := search-cost(curr-node, curr-node[lc-global])
edge:= get-edge(curr-node[lc-global])
cache-entry := get-lower-res(edge)
return cache-entry

End
4.6.3 IAA insert

Insertion of a new image object in the index has been performed as shown in the IAA insert algorithm. A top down search based on the image name has been carried out on the index tree. Binary search is performed at each level to find the needed key. If a corresponding IAA node entry is found, the new cache object is inserted into the IVG and least cost values are updated accordingly.

On the other hand, if no matching IAA node is found, a new entry is added at the right leaf level node and the index is redistributed in case the node size exceeds the order constraints. In case a new $LC$ value is arrived, it propagates in a bottom up manner as long as it does not exceed the $LC_{global}$ value of the visited IAA node.

**Algorithm: IAA insert**

input: cache-entry, tree

output: true or false

---

**Function IAA insert ()**

if(tree[root] = null)

then return false //cache-empty

end if

curr-node := tree[root]

while(curr-node != leaf-node)

curr-node := binary-search(curr-node, img-id[name])

//returns pointer to node corresponding to name

end loop

ivg-ref := binary-search(curr-node, img-id[name])
if(ivg-ref != null)
then insert(cache-entry, ivg-ref)
else ivg-ref := create-ivg(cache-entry)
end if

add-node(cache-entry[name], ivg-ref, curr-node)
if(is-full(curr-node))
then redistribute(tree)
end if

insert(cache-entry, ivg-ref)
if(cache-overflow)
then iaa-delete(cache-entry[size], tree)
end if

update-lc-values()
update-cache-state()
return true

End

4.6.4 IAA delete

In case of cache overflow, due to the lack of space for accommodating new incoming objects, the existing objects in the cache have been removed according to their LC value until enough space has been obtained. The tree structure has been reorganized to remove the least referenced object and making space for new objects to get inserted. When the number of leaf nodes in a level increases, it is split into two node structures with similar objects referenced closer to each other in each node.
Algorithm: IAA delete

input: size-to-free, tree
output: null

Function IAA delete ()

freed-size := 0

while(freed-size < size-to-free)

entry := iaa-search-by-cost(tree)

freed-size := freed-size + entry[size]

delete-object(entry)

end loop

End

4.7 CACHE MANAGEMENT MODULE

The structure of the cache management system has been systematically segmented into independent modules as shown in Figure 4.7 in order to perform the required operations.

The major components of the system include the content attribute extractor, IAA tree management, cache engine, and query strobe. These modules interact with the Adaptation Engine and the content server through communication channels whenever required, to retrieve multimedia contents from the servers. The individual modules are elucidated in the following section.
4.7.1 Content Attribute Extraction

Attribute extraction module reads the content information as and when new contents have been included in the cache. Parameters have been extracted including image size, resolution and bits per pixel. It supplies the essential parameters required for creating and maintaining the index.

4.7.2 IAA Tree Management

IAA Tree Management has been responsible for creating and maintaining cache index for optimal performance. It performs insertion, deletion and search operations on the IAA Tree to control the index structure. It takes actions based on image information received from the Query Strobe module. IAA Tree Management includes the Image Version Graph (IVG). The image version graph is mainly a clustering tool and has been used to organize the various related versions of an image.

The tree management operation includes maintaining the state of the system by dynamically assessing the appropriateness of the tree structure. Whenever the cache is full, a request will be received to compact the cache and at which the cache replacement algorithm has to be performed for removing the least referenced value and inserting the new object at the suitable node within the tree structure.
4.7.3 Cache Engine

The cache engine holds a repository of most relevant adapted images on the Content Proxy. It provides image references for attribute extraction. In addition, it imports newly adapted images into the cache. The objects in the cache possess a unique index and a value associated with its least cost reference. The cache engine utilizes the space allocated for storing the objects. Whenever the cache exceeds the limit, appropriate function for maintaining the cache limit is invoked. The cache size is regularized by the Cache Replacement Algorithm which traverses the tree structure and discards the objects with least cost value.

In case of an object addition in the cache, the tree has to be traversed to identify the suitable leaf node at which the new object can be added. After insertion of the object, reference of the root node to this object will be referenced along with the least cost value update. Whenever the size of the leaf node exceeds, tree splitting occurs and the structure has been divided into another tree structure, with a set of similar objects under another root node. The change has been intimated to other referencing nodes in the tree structure. Thus the cache has been maintained dynamically, to include new objects and release the least referenced objects.

4.7.4 Retrieval and Adaptation

Content Servers hold a database of the entire image repository. It provides images to Content Proxy for caching and determines the desired attributes for resultant adapted images. The request from the query processor invokes a search operation in the tree structure for the availability of a similar object. If the exact object has been identified, then it has been retrieved and delivered to the user.
When a similar transcodable object has been found, which is similar to the queried object but needs some adaptation in some of the parameters like size, resolution, etc., then the retrieved object has been provided to the content adaptation module for performing the necessary adaptation operations on the object and delivers the suitable adapted content to the user.

4.7.5 Query Processor

The user request has been received by the query strobe module. The request has been segregated into the request object, type of device, device capabilities, browser type, network bandwidth and related parameters. A streamlined request has been provided to the proxy cache module to retrieve the requested object. It provides information about the type of the required content and its associated parameters. When the requested object has been retrieved from the content proxy, the adapted object has been delivered to the user.

Query processor routes the request to the adaptation engine, when a suitable object is not found in the local cache of the content proxy. The parsed request from the user containing the requested object and the device characteristics has been provided to the adaptation engine. The resultant adapted content received from the adaptation engine has been delivered to the user device after updating in the cache of the content proxy for future retrieval.

4.8 PERFORMANCE EVALUATION

Performance analysis of the system was carried to ascertain the efficiency over other schemes. By comparing the performance of various architectures, the system can be evaluated for its performance.
4.8.1 Complexity Analysis

The search time complexity required by IAA tree has been evaluated and calculated as mentioned below. Consider the following parameters.

- \( h \) - number of levels in the tree
- \( m \) - order of the IAA tree
- \( i \) - number of nodes in the IAA tree, where \( n = \text{power}(m, h)-1 \)

Therefore, height of the IAA tree is given as \( \log_m n \). Choosing a branch in a node will take \( \log_2 m \), by employing binary search at each IAA node that is traversed in a search operation.

From the above parameters the search time complexity of IAA tree is deduced to be as mentioned in the Equation (4.12)

\[
O(\log_m n + h \log_2 m + V + E) \tag{4.12}
\]

where \( \log_m n \) constitutes the height of the IAA tree and \( \log_2 m \) is the time required to search in a single IAA node to determine which branch should be taken. Image names are stored in a node in ascending order that enables the use of binary search. Additional parameters were introduced because of the IVGs in use, viz. \( V \), the average number of vertices in an IVG node and \( E \), average number of edges in an IVG node.

Similarly, the delete time complexity is deduced to be using Equation (5.3).

\[
O(\log_m n + h m + V + E ) \tag{4.13}
\]
where \( m \) is the search time in a single IAA node to determine which branch should be taken. The appropriate branch will be determined based on the least cost which is unordered and therefore linear search needs to be employed. This explains the presence of factor \( h m \) in the delete time complexity.

### 4.8.2 Comparison of Indexing Efficiency

The performance of the IAA indexing structure can be effectively ascertained by comparing the results of a sample of cache searches. Conventional cache indexing architectures align either to the frequency of content being referenced or the timestamp as to when the content has been referenced previously. The performance results with a fixed cache size at 500 MB and the hit ratio sampled after every 100 image object queries has been analyzed and depicted in Figure 4.8.

![Figure 4.8 Performance for Cache Size of 500 MB](image)

The performance with a fixed cache size at 5 GB was analyzed and the hit ratio sampled after every 100 image object queries has been provided in the graph (Figure 4.9).
Figure 4.9 Performance for Cache Size of 5 GB

The simulation results depicted in (Figure 4.8) and (Figure 4.9) indicate that the performance of the IAA Indexing scheme has been increased with higher cache sizes and attained a higher saturation value compared to the Frequency based and Timestamp based schemes. For a comparatively smaller cache size of 500 MB, the hit ratio decreases when the cache overflow scenario occurs. For higher cache sizes like 5 GB, IAA scheme has shown a better saturation value for hit ratio when compared to the other two schemes that do not take adaptation parameters into account.

4.9 SUMMARY

The proxy server introduced between the user and the server has reduced the load on the server by servicing the user requests. The cache available in the proxy server helps to store and retrieve frequently accessed objects easily without sending requests to content servers. The cache has been managed suitably to maintain the cache size and presence of frequently referenced objects. Dedicated cache management module ensures that the cache has been maintained appropriately.
The indexing mechanism implemented in the proxy cache is adaptation aware and hence related objects are stored closer. The indexing structure helps to store the objects in a tree structure and ensures that the objects have been retrieved efficiently. Similar transcodable objects have been indexed closer, so that they can be easily be adapted and the required object characteristics can be obtained.

Overall system performance has been improved by 30% through the introduction of the proxy server containing appropriate context aware indexing mechanism with a suitable cache. The load on the server has been reduced by servicing the request from the user. As similar objects indexed nearer, retrieval of the objects become faster. Thus the proposed proxy server handles the request from the user and provides the adapted content within minimal time to enhance the user satisfaction.