CSE 430: Operating Systems

Instructor: Dr. Violet R. Syrotiuk

Adaptive Replacement Cache Policy

Design and Analysis Report

Project 3: Final

Author: Ankit Rathi

ASU ID: 1207543476

Contents

ARC Caching Scheme

Adaptive Replacement Cache (ARC) is a page replacement algorithm with better performance than LRU. ARC makes use of two lists, one to maintain the “Most Recently Used – T1” and “Most Frequently Used – T2” lists. Each of these list is extended to another list “Most Recently Used Ghost – B1” and “Most Frequently Used Ghost – B2”. Whenever an element is evicted from T1, it is moved to B1 and element evicted from T2 is moved to B2. T1 and B1 together can be referred by L1 and T2 and B2 together can be referred by L2.

Data Structure to implement cache

I have implemented the cache with the help of cache data structure. The cache data structure consists of 4 LRU lists/queues. I have implemented the Queue using a Doubly Linked List. Doubly Linked List is a dynamic data structure that can grow size of the number of pages that can be accommodated completely in the cache. Each node of the queue is implemented with the help of another data structure that is QNode. QNode can contains pointers to point to the previous and the next element in the queue. The maximum size of the queue will be equal to the total number of frames available. One more important thing to be noted is that most recently used page will at the front of the queue and the least recently used pages will be at the rear end of the queue. Initial assumption that has been made is that initially no pages are present in the cache.

Each node of the queue is represented using the QNode structure. This structure contains the pointers to the previous and next nodes in the doubly linked list. It also contains a variable to hold the page number. This stores the values of the page number that has been accessed.

```
/*
* A Queue Node (Implementation of queue using Doubly Linked List)
*/
typedef struct QNode
{
	struct QNode *prev, *next;
	unsigned int iPageNumber;  // the page number stored in this QNode
} QNode;
```

The below data structure represents the Queue structure. The queue structure contains a count variable to keep track of the number of frames that have been filled in the cache. It also contains a variable to keep note of the total number of frames or pages that can be loaded into the memory at the same time. It also keeps tracks of the front and rear end of the queue. This is required as we insert the elements in the front end and but remove the elements from the rear end. This is a basic implementation of the double linked list which provide the features of a FIFO queue.

```
/*
*A FIFO Queue using QNodes
*/
typedef struct Queue
{
	unsigned int iCount;  // Number of frames that have been filled.
```

```
        unsigned int iNumberOfFrames; // total number of frames
        QNode *front, *rear;
} Queue;
```

Struct ARCCache is used to represent the entire cache structure. It consists of variable to hold the cache size and an adaptation parameter. Also it has 4 instances of LRU lists.

```
/*
*A ARC Cache consisting of 4 Queues
* mrug (B1)- Most Recently Used Ghost
* mru (T1) - Most Recently Used
* mfu (T2) - Most Frequently Used
* mfug (B2) - Most Frequently Used Ghost
*/
struct ARCCache
{
        unsigned int c;
        float p;
        struct Queue mrug, mru, mfu, mfug;
};
```

newQNode is a utility function to create a new node. This node will be used to store the pagenumber that has been accessed.

```
/*
*A utility function to create a new Queue Node. The queue Node
* will store the given 'pageNumber'
*/
QNode* newQNode(unsigned int pageNumber)
{
        // Allocate memory and assign 'pageNumber' for the first cache
        QNode* temp = (QNode *)malloc(sizeof(QNode));
        temp->iPageNumber = pageNumber;

        // Initialize prev and next as NULL
        temp->prev = NULL;
        temp->next = NULL;

        return temp;
}
```

createQueue is a utility function to create an empty queue. The queue can at most accommodate the numberofframe variable of the data structure. This function gets called only once during the entire program execution. So for the first time all the initializations needs to be done. Front and the rear pointers are initialized to NULL.

```
/*
* A utility function to create an empty Queue.
* The queue can have at most 'numberOfFrames' nodes
*/
Queue* createQueue(unsigned int numberOfFrames)
{
        Queue* queue = (Queue *)malloc(sizeof(Queue));
```

```
	// The queue is empty
	queue->iCount = 0;
	queue->front = NULL;
	queue->rear = NULL;

	// Number of frames that can be stored in memory
	queue->iNumberOfFrames = numberOfFrames;

	return queue;
}
```

isQueueFull function is used to check if the number of pages in the cache has reached its maximum limit.

```
/*
*A function to check if there is slot available to bring
* in a new page into memory.
*/
int isQueueFull(Queue* queue)
{
	return queue->iCount == queue->iNumberOfFrames;
}
```

isQueueEmpty function is used to check if the queue is empty.

```
/*
*A function to check if queue is empty
*/
int isQueueEmpty(Queue* queue)
{
	return queue->rear == NULL;
}
```

deQueue function is used to delete a page from queue. If the queue is empty then the function just returns back. It performs the delete operation based on whether the node that it has received as input is the only node in the queue, whether it is front end of the queue, whether it is rear end of the queue or whether the queue is the not at the front/rear position. After the delete operation, pointers are updated accordingly. Then the counter is decremented by one.

```
/*
*A utility function to delete a frame from queue
*/
void dequeueNode(struct Queue* queue, struct QNode* reqPage)
{
	/* base case */
	if (reqPage == NULL)
	{
		printf("Requested Page is Not Available\n");
		return;
	}
	/*If the node to be removed is the only node in the queue*/
	if ((queue->front == reqPage) && (queue->rear == reqPage))
```

```
	{
		queue->front = NULL;
		queue->rear = NULL;
	}
	/*If the node to be removed is the node at the front of the queue/list*/
	else if ((queue->front == reqPage) && (queue->rear != reqPage))
	{
		queue->front = reqPage->next;
		queue->front->prev = NULL;
	}
	/*If the node to be removed is the node at the end of the queue/list*/
	else if ((queue->front != reqPage) && (queue->rear == reqPage))
	{
		queue->rear = reqPage->prev;
		queue->rear->next = NULL;
	}
	/*If the node to be removed is the node in the some other location other than
front and rear end*/
	else
	{
		reqPage->prev->next = reqPage->next;
		reqPage->next->prev = reqPage->prev;
	}

	/*Free the node after dequeuing it from the queue*/
	free(reqPage);

	// decrement the number of full frames by 1
	queue->iCount--;
}
```

enqueueMRU function is used to add the element into the queue at the MRU position. The function first creates a new node to be created with the given page number. Then it checks if the queue is empty, then update the front and rear pointers. If the queue is not empty then only the front pointers are updated. The count variable is incremented by one.

```
/*
*A function to add a page with given 'pageNumber' to queue.
*/
void enqueueMRU(struct Queue* queue, unsigned int pageNumber)
{
	/*Create a new node with given page number, And add the new node to the front of
queue*/
	struct QNode* temp = newQNode(pageNumber);
	temp->next = queue->front;

	/*If queue is empty, change both front and rear pointers*/
	if (isQueueEmpty(queue))
	{
		queue->front = temp;
		queue->rear = queue->front;
	}
	/*Else change the front*/
	else
	{
```

```
                queue->front->prev = temp;
                queue->front = temp;
        }

        /*increment number of full frames*/
        queue->iCount++;
}
```

ReferencePage function is used to reference a page from the cache memory. There are two cases that are handled by the function. First, If the frame is not in memory, then NULL is returned back by the function. In case the page is available in memory, then in such a situation, the pointer to the page location is passed back to the callee.

```
/*
* This function is called when a page with given 'pageNumber' is referenced
* from cache (or memory). There are two cases:
* 1. Frame is not there in memory, then it returns NULL
* 2. Frame is there in memory, then return the pointer to the node
*/
struct QNode* ReferencePage(struct Queue* queue, unsigned int iPageNumber)
{
        struct QNode* reqPage = NULL;

        /*Check if the page or the frame being referenced is already present in the queue
of nodes.*/
        struct QNode* tempReqPage = NULL;
        for (tempReqPage = queue->front; tempReqPage; tempReqPage = tempReqPage->next)
        {
                if (tempReqPage->iPageNumber == iPageNumber)
                {
                        reqPage = tempReqPage;
                        return reqPage;
                }
        }
        return reqPage;
}
```

moveXToT2 function is used to move an element from one queue to another queue. It also takes input as the node that needs to be moved. First part of the function is used to remove the element from the queue. The element may be present at any location, front, rear, or in the middle. All these conditions are checked and then the element is moved from the list to top of T2. One count is decremented from the queue from which the page is removed and one count is incremented from the queue into which the element is added.

```
/*
* This function is called when a page with given 'pageNumber' is to be
* moved from one queue to another queue. It takes input of the queue from which
* the page needs to be moved and queue into which the page needs to be inserted.
* Third input the function is the page that needs to be moved.
*/
void moveXToT2(struct Queue* fromQueue, struct Queue* toQueue, struct QNode* reqPage)
{
        /*Remove the requested page from fromQueue Check if this the only element in the
fromQueue*/
        if ((fromQueue->front == reqPage) && (fromQueue->rear == reqPage))
```

```
	{
		fromQueue->front = NULL;
		fromQueue->rear = NULL;
	}
	/*Remove the requested page from fromQueue, Check if this the first element in the
fromQueue*/
	else if ((fromQueue->front == reqPage) && (fromQueue->rear != reqPage))
	{
		fromQueue->front = reqPage->next;
		fromQueue->front->prev = NULL;
	}
	/*Remove the requested page from fromQueue, Check if this the last element in the
fromQueue*/
	else if ((fromQueue->front != reqPage) && (fromQueue->rear == reqPage))
	{
		fromQueue->rear = reqPage->prev;
		fromQueue->rear->next = NULL;
	}
	/*Remove the requested page from fromQueue, Check if this is not the first/last
element in the fromQueue*/
	else
	{
		reqPage->prev->next = reqPage->next;
		reqPage->next->prev = reqPage->prev;
	}
	/*decrement the number of pages in fromQueue by 1*/
	fromQueue->iCount--;


	reqPage->next = toQueue->front;

	/*If queue is empty, change both front and rear pointers*/
	if (isQueueEmpty(toQueue))
	{
		toQueue->front = reqPage;
		toQueue->rear = toQueue->front;
	}
	/*Else change the front*/
	else
	{
		toQueue->front->prev = reqPage;
		toQueue->front = reqPage;
	}

	/*Increment the number of pages in toQueue by 1.*/
	toQueue->iCount++;
}
```

replace function is called to move rear element from T1 to B1 and T2 to B2. This function is implemented as part of the algorithm provide in the question description.

```
/*
* This function is called when a page with given 'pageNumber' is to be moved from
* T2 to B2 or T1 to B1. Basically this function is used to move the elements out from
* one list and add it to the another list beginning.
*/
void replace(const unsigned int iPageNumber, const float p)
```

```
{
	if ((cache.mru.iCount >= 1) && ((cache.mru.iCount > p) || ((NULL !=
ReferencePage(&cache.mfug, iPageNumber)) && (p == cache.mru.iCount))))
	{
		if (cache.mru.rear)
		{
			moveXToT2(&cache.mru, &cache.mrug, cache.mru.rear);
		}
	}
	else
	{
		if (cache.mfu.rear)
		{
			moveXToT2(&cache.mfu, &cache.mfug, cache.mfu.rear);
		}
	}
}
```

Arc_lookup function is the most important function of the program. It contains the main business logic to carry out the ARC cache implementation. It takes care of the 4 cases as described in the algorithm, case1: Page is found in T1 or T2, then it is a hit. Case2: Page is found in B1, then it is miss and then page needs to be added to the top of the T2 list. Case3: Page is found in B2, then it is a miss and the page needs to be added to top of the T2 list. Case4: This is the case when the page is not found in all the four list, that means it is a miss and the element needs to be added to the T1 list. All the cases have been implemented with the conditions as is specified in the algorithm. The adaptation parameter p is updated in case of Case2 and Case3. In case there is hit in B1, then T1 size is increased, meaning p is increased and in case there is a hit in B2, then T2 size is increased, thereby p is decreased.

```
/*
* This is the function that is used to Lookup an object with the given key.
* It consists of 4 cases, each case represents the list to which the element
* or the given page may belong.
*/
void arc_lookup(const unsigned int iKeyPageNumber)
{
	struct QNode* reqPage = NULL;
	if (hash[iKeyPageNumber%HASHSIZE] > 0)
	{
	/*Case 1: Part A: Page Found in MRU (T1)*/
	if (NULL != (reqPage = ReferencePage(&cache.mru, iKeyPageNumber)))
	{
#if PRINTDEBUGS == 1
		//printf("HIT :: Case 1: Part A: Page Found in MRU (T1)\n");
#endif
		/*Increment the hit counter*/
		iHitCount++;

		/*Move xt to MRU position in T2.*/
		moveXToT2(&cache.mru, &cache.mfu, reqPage);
	}
	/*Case 1: Part B: Page Found in MFU (T2)*/
	else if (NULL != (reqPage = ReferencePage(&cache.mfu, iKeyPageNumber)))
	{
#if PRINTDEBUGS == 1
		//printf("HIT :: Case 1: Part B: Page Found in MFU (T2)\n");
```

```
#endif
                /*Increment the hit counter*/
                iHitCount++;

                /*Move xt to MRU position in T2.*/
                moveXToT2(&cache.mfu, &cache.mfu, reqPage);
        }
        /*Case 2: Page Found in MRUG (B1).*/
        else if (NULL != (reqPage = ReferencePage(&cache.mrug, iKeyPageNumber)))
        {
#if PRINTDEBUGS == 1
                //printf("MISS :: Case 2: Page Found in MRUG (B1).\n");
#endif
                iMissCount++;
                cache.p = (float)MIN((float)cache.c, (cache.p + MAX((cache.mfug.iCount*1.0)
/ cache.mrug.iCount, 1.0)));
                replace(iKeyPageNumber, cache.p);

                /*Move xt from B1 to the MRU position in T2.*/
                moveXToT2(&cache.mrug, &cache.mfu, reqPage);
        }
        /*Case 3: Page Found in MFUG (B2).*/
        else if (NULL != (reqPage = ReferencePage(&cache.mfug, iKeyPageNumber)))
        {
#if PRINTDEBUGS == 1
                //printf("MISS :: Case 3: Page Found in MFUG (B2).\n");
#endif
                iMissCount++;
                cache.p = (float)MAX(0.0, (float)(cache.p - MAX((cache.mrug.iCount*1.0) /
cache.mfug.iCount, 1.0)));
                replace(iKeyPageNumber, cache.p);

                /*Move xt from B2 to the MRU position in T2.*/
                moveXToT2(&cache.mfug, &cache.mfu, reqPage);
        }
        /*Case 4: Page Not Found in MRU, MRUG, MFU, MFUG.*/
        else
        {
#if PRINTDEBUGS == 1
                //printf("Case 4: Page Not Found in MRU, MRUG, MFU, MFUG.\n");
#endif
                iMissCount++;
                /*Case 4: Part A: L1 has c pages*/
                if ((cache.mru.iCount + cache.mrug.iCount) == cache.c)
                {
                        if (cache.mru.iCount < cache.c)
                        {
                                hash[cache.mrug.rear->iPageNumber % HASHSIZE]--;

                                /*Delete LRU page in B1.*/
                                dequeueNode(&cache.mrug, cache.mrug.rear);
                                replace(iKeyPageNumber, cache.p);
                        }
                        else
                        {
                                /*Here B1 is empty. Delete LRU page in T1 (also remove it from
the cache).*/
                                hash[cache.mru.rear->iPageNumber % HASHSIZE]--;
```

```
                        /*Delete LRU page in T1.*/
                        dequeueNode(&cache.mru, cache.mru.rear);
                    }
                }
                /*Case 4: Part B: L1 has less than c pages*/
                else if ((cache.mru.iCount + cache.mrug.iCount) < cache.c)
                {
                    if ((cache.mru.iCount + cache.mfu.iCount + cache.mrug.iCount +
cache.mfug.iCount) >= cache.c)
                    {
                        if ((cache.mru.iCount + cache.mfu.iCount + cache.mrug.iCount +
cache.mfug.iCount) == (2 * cache.c))
                        {
                            hash[cache.mfug.rear->iPageNumber % HASHSIZE]--;

                            /*Delete LRU page in B2.*/
                            dequeueNode(&cache.mfug, cache.mfug.rear);

                        }
                        replace(iKeyPageNumber, cache.p);
                    }
                }
                /*Fetch xt to the cache and move it to MRU position in T1.*/
                enqueueMRU(&cache.mru, iKeyPageNumber);
                hash[iKeyPageNumber % HASHSIZE]++;
            }
            }
            else
            {
#if PRINTDEBUGS == 1
                //printf("Case 4: Page Not Found in MRU, MRUG, MFU, MFUG.\n");
#endif
                iMissCount++;
                /*Case 4: Part A: L1 has c pages*/
                if ((cache.mru.iCount + cache.mrug.iCount) == cache.c)
                {
                    if (cache.mru.iCount < cache.c)
                    {
                        hash[cache.mrug.rear->iPageNumber % HASHSIZE]--;

                        /*Delete LRU page in B1.*/
                        dequeueNode(&cache.mrug, cache.mrug.rear);
                        replace(iKeyPageNumber, cache.p);
                    }
                    else
                    {
                        /*Here B1 is empty. Delete LRU page in T1 (also remove it from
the cache).*/
                        hash[cache.mru.rear->iPageNumber % HASHSIZE]--;

                        /*Delete LRU page in T1.*/
                        dequeueNode(&cache.mru, cache.mru.rear);
                    }
                }
                /*Case 4: Part B: L1 has less than c pages*/
                else if ((cache.mru.iCount + cache.mrug.iCount) < cache.c)
                {
```

```
                    if ((cache.mru.iCount + cache.mfu.iCount + cache.mrug.iCount +
cache.mfug.iCount) >= cache.c)
                    {
                            if ((cache.mru.iCount + cache.mfu.iCount + cache.mrug.iCount +
cache.mfug.iCount) == (2 * cache.c))
                            {
                                    hash[cache.mfug.rear->iPageNumber % HASHSIZE]--;

                                    /*Delete LRU page in B2.*/
                                    dequeueNode(&cache.mfug, cache.mfug.rear);
                            }
                            replace(iKeyPageNumber, cache.p);
                    }
            }

            /*Fetch xt to the cache and move it to MRU position in T1.*/
            enqueueMRU(&cache.mru, iKeyPageNumber);
            hash[iKeyPageNumber % HASHSIZE]++;
      }
}
```

arc_queue_init function is called to initialize the values of variable of structure of the queue. Count is set to 0, front and rear are set to NULL.

```
/*
* This function is called to initialize the values of variable of structure of the queue.
* Count is set to 0, front and rear are set to NULL.
*/
void arc_queue_init(struct Queue *queue, unsigned int numberOfFrames, char *name)
{
      queue->name = name;
      // The queue is empty
      queue->iCount = 0;
      queue->front = NULL;
      queue->rear = NULL;

      // Number of frames that can be stored in memory
      queue->iNumberOfFrames = numberOfFrames;
}
```

initARCCache function is called to initialize the cache structure. In the structure, the cache size is set, the adaptation parameter p is set to 0. Calls to function arc_queue_init is made to initialize all the queues as part of the structure of cache.

```
/*
* This function is called to initialize the cache structure.
* In the structure, the cache size is set, the adaptation page is set to 0.
* and then a call to function is made to initialize all the queues as part of
* the structure of cache.
*/
void initARCCache(const unsigned int iCacheSize)
{
      cache.c = iCacheSize;
      cache.p = 0;

      char *mrug = "MRUG";
      char *mru = "MRU";
```

```
	char *mfu = "MFU";
	char *mfug = "MFUG";

	/*Initialize the queue for each of the list*/
	arc_queue_init(&cache.mrug, iCacheSize, mrug);
	arc_queue_init(&cache.mru, iCacheSize, mru);
	arc_queue_init(&cache.mfu, iCacheSize, mfu);
	arc_queue_init(&cache.mfug, iCacheSize, mfug);
}
```

Main functions first reads the arguments supplied along with the file. If less than 2 arguments are supplied then an error will be generated. After that I have created and initialized the queue. Then I have opened the file using the fopen command and read line by line the contents of the file. After reading each line I reference that page in the memory. This is done in infinite loop till the end of the file is reached. Then the hit ratio is calculated.

```
/*
*Driver program to test above functions
*/
int main(int argc, char **argv)
{
	FILE *fp;
	time_t start, stop;

	if (argc == 3)
	{
		/*argv[1] is the cache size*/
		/*argv[2] is the Trace File Name*/
		iCacheSize = strtol(argv[1], NULL, 10);
		fileName = argv[2];
	}
	else
	{
		printf("Cache Size and Trace File Name not supplied properly.\nProgram
Terminating\n");
		getchar();
		exit(-1);
	}

	printf("Cache Size = %d\n", iCacheSize);
	printf("TraceFile Name = %s\n", fileName);

	/*Initialize cache can hold c pages*/
	initARCCache(iCacheSize);

	fp = fopen(fileName, "r");
	if (fp == NULL)
	{
		printf("Error while opening the file: %s\n", fileName);
		getchar();
		exit(EXIT_FAILURE);
	}

	unsigned int iStartingBlock = 0, iNumberOfBlocks = 0, iIgnore = 0, iRequestNumber
= 0;
	unsigned int i = 0;
	unsigned int iTotalRequests = 0;
```

```
	time(&start);

	while (1)
	{
		if (-1 != fscanf(fp, "%u %u %u %u", &iStartingBlock, &iNumberOfBlocks,
&iIgnore, &iRequestNumber))
		{
			//printf("%u-->%u\n", iRequestNumber, cache.mru.iCount);
#if PRINTDEBUGS == 1
			//printf("%u\n", iRequestNumber);
#endif
			//printf("%u\t\t%u/%u\t\t%u/%u\t\t%u/%u\t\t%u/%u\n", iRequestNumber,
cache->mru.iCount, cache->mru.iNumberOfFrames, cache->mrug.iCount, cache-
>mrug.iNumberOfFrames, cache->mfu.iCount, cache->mfu.iNumberOfFrames, cache->mfug.iCount,
cache->mfug.iNumberOfFrames);
			for (i = iStartingBlock; i < (iStartingBlock + iNumberOfBlocks);
i++)
			{
				iTotalRequests++;
#if PRINTDEBUGS == 1

	printf("**********************************************************************\nRequested
Page = %u\n**********************************************************************\n", i);
#endif
				/*Call to check if page is in cache or not*/
				arc_lookup(i);
#if PRINTDEBUGS == 1
				printList();
#endif
			}
		}
		else
		{
			break;
		}
	}

	time(&stop);
	/*Print the important statistics about the result for cache hit and miss ratios*/
	printf("Miss Count = %u\n", iMissCount);
	printf("Hit Count = %u\n", iHitCount);
	//printf("iMissCount + iHitCount = %u\n", iMissCount + iHitCount);
	//printf("Total Requests = %u\n", iTotalRequests);
	printf("Hit Ratio = %5.4f %%\n", ((float)(iHitCount * 100) / (iHitCount +
iMissCount)));
	printf("Rounded Hit Ratio = %5.2f %%\n", floor(((float)(iHitCount * 100) /
(iHitCount + iMissCount)) * 100 + 0.5) / 100);
	printf("Finished in about %.0f seconds. \n", difftime(stop, start));

	getchar();
	return 0;
}
```

Representation of important variables

I have used unsigned int for counter variables to keep count of number of hits and number of misses. The cache size that I receive as an input from the user is also stored in an unsigned int.

```
/*
*Global Variable Declarations
*/
unsigned int iCacheSize = 0;
char *fileName;
unsigned int iMissCount = 0;
unsigned int iHitCount = 0;
unsigned int hash[HASHSIZE];
```

The elements of QNode are used as shown below. Page Number is stored as Unsigned int. The next and prev pointers are of type struct QNode.

```
/*
* A Queue Node (Implementation of queue using Doubly Linked List)
*/
struct QNode
{
	unsigned int iPageNumber;  /*the page number stored in this QNode*/
	struct QNode *next; /*Pointer to Next Node*/
	struct QNode *prev; /*Pointer to Previous Node*/
};
```

The elements of Queue are used as shown below. It contains iCount to store the count of number of pages available in memory for a Queue. Front and rear are the pointers to the struct QNode. Name is stored as character.

```
/*
*A FIFO Queue using QNodes
*/
struct Queue
{
	unsigned int iCount;  /*Number of frames that have been filled.*/
	unsigned int iNumberOfFrames; /*Total number of frames.*/
	struct QNode *front, *rear; /*Front and Rear Pointers*/
	char *name;   /*Name of the Queue like MRU, MFU, MRUG, MFUG*/
};
```

The elements of ARCCache are used as shown below. It uses unsigned int to store the cache size. The adaptation parameter is stored as float. The 4 queues T1(mru), T2(mfu), B1(mrug), B2(mfug) are used as type struct Queue.

```
/*
*A ARC Cache consisting of 4 Queues
* mrug (B1)- Most Recently Used Ghost
* mru (T1) - Most Recently Used
* mfu (T2) - Most Frequently Used
* mfug (B2) - Most Frequently Used Ghost
*/
struct ARCCache
{
	unsigned int c;
	float p;
	struct Queue mrug, mru, mfu, mfug;
};
```

The arc cache variable is declared as global and can be accessed by any function.

```
/*
*Global Declarations of cache structure
*/
struct ARCCache cache;
```

Hit Ratio

The Hit ratio is calculated as the ratio of number of hits to the total number of cache references. The below equation gives the hit ratio in percentage. The numerator is converted to float before division to auto typecasting happens and the result is displayed in flat form.

$$Hit\ Ratio = \frac{(float)(iHitCount * 100)}{iHitCount + iMissCount}$$

To Round of the Hit Ratio to the hundredth decimal, I have used the math.h library function floor. The floor function rounds of the value to the next correct decimal value by using the equation shown below.

$$Hit\ Ratio = floor\left(\frac{\left(\left(\frac{(float)(iHitCount * 100)}{iHitCount + iMissCount}\right) * 100 + 0.5\right)}{100}\right)$$

Improve Time Complexity

The program makes use of doubly linked list to implement the cache. Hence to search for an element in the queue it takes time of O(cache size). And the total number of pages that we reference be equal to the say NumberOfPageReference, there are 4 queues that we are maintaining, hence the complexity of the algorithm implemented can be considered to be O(4 * Cache Size * NumberOfPageReference).

To improve the scanning of element in the Linked List, I have made use of hash function. The hash function is updated whenever an element is added into any of the queue and hash function is cleared in case any of the page is removed from the cache. Since it is difficult to implement the hash with large size, I have used hash of size 1000000. Hash is declared as an array of size 1000000. This array is incremented whenever a page is bought into the cache and decremented whenever a page is removed from the cache.

```
#define HASHSIZE 1000000

unsigned int hash[HASHSIZE];
```

The hash function is implemented in such a way that returns a value to indicate that the element may be present in the cache without guarantee. But with guarantee it can say that the element is not present in the cache. When hash says that the element is not present in cache, then we can directly go to execute the case4 of the algorithm, instead of wasting time to traverse through the 4 lists to determine that the element is not present. The basic structure of the arc_lookup function is as shown below,

```
if (hash[iKeyPageNumber%HASHSIZE] > 0)
```

```
{
        Case 1
        Case 2
        Case 3
        Case 4
}
else
{
        Case 4
}
```

From the above code it can be seen that, in case the hash says element may be present, we follow the algorithm as mentioned in the question with all the 4 cases, but in case the hash says that the element is not present, then we can directly go to case 4. This saves us a lot of time in traversing through the lists.

Problems Faced

One problem that I faced is with respect to the using of unsigned int for p in the first case. Since I had made the variable as unsigned, the negative values during adaptation were making it too huge. This was resulting in wrong output. Took a very long time to debug, that MAX and MIN function were returning the wrong values due to int getting type casted into unsigned int.

Another problem that I faced was, the program was too slow in execution, and hence I had to come up with some way to reduce the time complexity. So to reduce the time complexity I introduced an array hash, which stores information about the page being present in the cache or not.

```
/*Fetch xt to the cache and move it to MRU position in T1.*/
enqueueMRU(&cache.mru, iKeyPageNumber);
hash[iKeyPageNumber % HASHSIZE]++;
```