CACHE PERFORMANCE ANALYSIS:
A TRACE-DRIVEN SIMULATION

By

PAMELA NEELAVENI

Bachelor of Engineering (Hons)

Birla Institute of Technology and Science

Rajasthan, INDIA

1990

Submitted to the faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirement for
the Degree of
MASTER OF SCIENCE
July 1994

OKLAHOMA STATE UNIVERSITY

CACHE PERFORMANCE ANALYSIS:
A TRACE-DRIVEN SIMULATION

Thesis Approved:

Mansur Samadzadeh
Thesis Advisor

Blayne E. Mayfield

Mitchell Neilsen

Thomas C. Collins
Dean of the Graduate College

ACKNOWLEDGEMENTS

I thank my graduate advisor Dr. Mansur H. Samadzadeh for his advice, assistance, and guidance. His constructive criticism helped me gain confidence. During my whole graduate studies, I got inspiration and motivation due to his constant guidance. My sincere thanks to Drs. Blayne Mayfield and Mitch Neilsen for serving on my graduate committee.

I also want to thank Mr Jim McGee and Mr. Andy Adsit, my supervisor at the University Computer Center, OSU, for allowing flexible working hours.

I would like to thank my husband Srikanth for his strong encouragement at times of difficulty, love and understanding throughout this whole process. Finally, I would like to express my gratitude to my parents, brother, and sisters. Without their support and encouragement, this task would not have been possible.

TABLE OF CONTENTS

Chapter Page

LIST OF FIGURES

Figure Page

LIST OF TABLES

CHAPTER I

INTRODUCTION

Cache memory is used in most computer systems. An important goal in the design of a computer system is that it should behave according to the expectations of the designer. The performance of a system can be captured and evaluated using various techniques. Trace-driven simulation is one of the techniques used to study the performance of a computer system.

Nowadays most of the small, medium, and large machines have cache memories to improve their performance. Information located in cache can generally be retrieved in less time than the information located in main memory [Smith82]. Trace-driven simulation is a technique by which, using the actual address traces as the external stimuli, a model of a proposed system, e.g., cache memory, can be evaluated.

Several address tracing techniques have been developed over the last ten years, each one with its own merits and demerits [Stunkel91]. These techniques are typically analyzed with respect to issues such as speed, flexibility, completeness, reduction in execution time, and accuracy. Different methods of address tracing techniques can be classified into five categories as given in Figure 1 (adapted from [Stunkel91]). A brief description of these five techniques is given below.

In the hardware monitored technique, the address traces are directly recorded

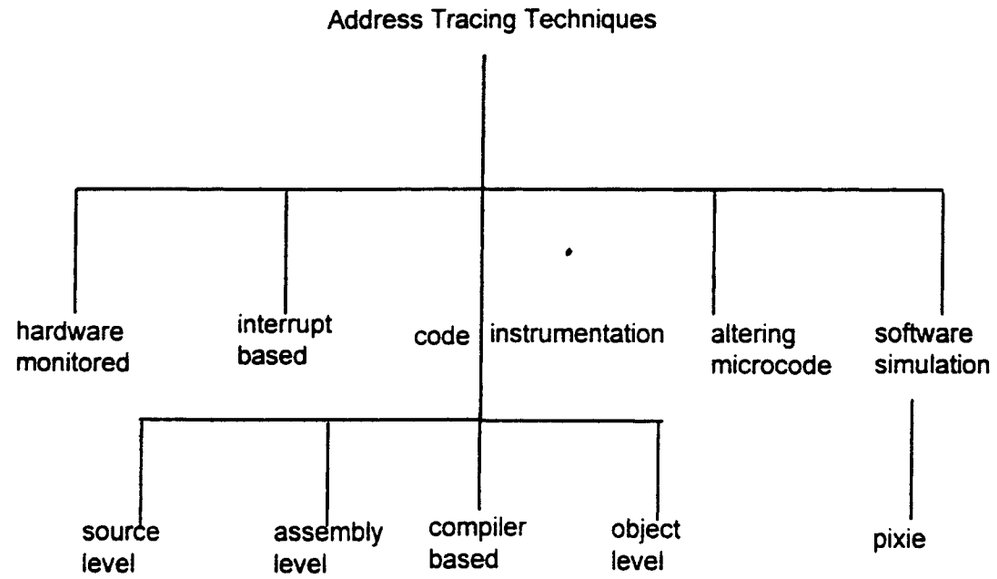


Figure 1. Different address tracing techniques

off processor memory requests when they are sent to off-chip caches and memory chips. In the altering microcode technique, commonly known as ATUM (address tracing using microcode), the traces are obtained by making minor changes to the existing microcode of a machine. This technique has been employed to obtain addresses for VAX architectures [Agarwal88]. In the interrupt based technique, every instruction generates a CPU interrupt and the interrupt routine analyzes the opcode, calculates the memory addresses, if any, and stores it in a buffer. Most architectures provide a trap bit that can be enabled and a corresponding interrupt routine that can be modified to acquire the traces. In the instrumented program technique, the application program is instrumented at specific points. During run time, these extra instructions log the trace information which, when postprocessed, gives the actual trace. The code level insertion technique can be carried out at various levels such as source code level, assembly level, or binary level

or object level [Stunkel91]. Software simulation methods can model processor execution and simultaneously provide user traces. Pixie is one of the software simulation methods used to capture traces. This method of generating address traces was initially developed on SPARC systems at Berkeley [Lovett93].

The most accurate way of studying cache performance, before a machine is actually built, is through simulation [Marcovitz88]. By changing the parameters of a simulation model, it is possible to simulate a cache of any size. Using this kind of approach and model, one can design a cache model for a required behavior. If some discrepancies are detected, based on the performance analysis of the model, the cache can be redesigned.

Chapter II of this thesis provides a review of the current literature on the trace-driven simulation technique and memory management in general. Chapter III provides a discussion on the design and the implementation details of the software that was developed as part of this thesis. The testing and evaluation of the software developed are discussed in Chapter IV. The last chapter, Chapter V, provides a summary of this thesis, the conclusions drawn from this study, and the suggestions for future work.

CHAPTER II

LITERATURE REVIEW

2.1 Introduction

The most accurate method of determining the performance of a specific computer design or the validity of a new architectural approach, is to build it [Lilja93]. A complete implementation is time consuming and expensive, and generally precludes the opportunity for using the performance evaluation for tuning the system. Therefore, it is necessary to explore the details of the design, before building a system, using mathematical analysis or by simulation. A primary goal in modeling a system before constructing the actual system is to reduce the memory access time in order to reduce the execution time and improve the performance of the system. Since cache memories are often used in modern computer systems, the study of cache size, mapping, and replacement algorithms is an important field in computer system performance evaluation.

2.2 Definitions

This section contains some of the basic definitions about cache memory that are used in this thesis. These definitions are mostly based on three major references [Smith82] [Agarwal88] [Marcovitz88].

• A *trace* is an address sequence obtained by executing a program and recording every memory location referenced by the program during its execution.

• *Locality of reference* is a property exhibited by running processes, that processes tend to reference storage in nonuniform, highly localized patterns.

• All data that is written by at least one processor, and read or written by at least one other processor, is marked as *non-cacheable.*

• *Clumpiness* means occurring close together. In this thesis, misses refer to cache misses. Clumpiness in misses refers to misses occurring close, or almost overlapping.

• *Prefetch* is to get data or instructions required by a program before they are actually needed.

• *Block size* or *line size* is the amount of storage associated with an address tag.

• A *cache miss* in a cache occurs whenever the desired information is not available in the cache.

• A *cache hit* in a cache occurs whenever the desired information is available in the cache and the processor does not have to wait for the information.

• A *block* is defined as a group of words which can be read from or written to a device. A *block* in a cache can be divided into words. A block can have any number of words. Whenever there is a miss, instead of getting one word, a whole block is brought into the cache.

• When the CPU executes instructions that modify the contents of the current address space, those changes must be reflected in main memory. Effecting the modifications immediately to the main memory is called *write-through.*

· When the CPU executes instructions that modify the contents of the current address space, those changes can be initially modified in cache and later be reflected in the memory. This is called *copy-back.*

· *Page map table* is a table used to map virtual addresses onto physical addresses.

· *Multiprogramming* is defined as a collection of processes running logically in parallel where the CPU switches from one process to another process.

· When more than one process is requesting the CPU, the operating system must decide which one to run first. That part of the operating system concerned with this decision is called the scheduler and the process of assigning the CPU to jobs is called *scheduling.*

2.3 Storage Hierarchy

Storage hierarchy refers to arranging storage devices on the basis of access speed and cost so that only the most important information, i.e., the programs and data referenced by the CPU directly, is kept on the expensive fast devices and the rest of the information is kept on inexpensive slow devices [Leung82]. The principal reason in having a hierarchial memory system is to improve the effective memory access time and accordingly increase the processing speed [Smith82]. For example, in a two-level memory hierarchy system having a main memory and an auxiliary memory, the information must first be moved to primary storage before it can be referenced by the CPU. Thus the auxiliary memory has a copy of all the information stored in main memory. When a copy of data is modified in main memory, the copy of data in auxiliary memory must also be modified using a write-through or copy-back scheme. In a two-level system, the data is

referenced from the main memory. If the data that is referenced is not available in the main memory, then the data must be transferred from the auxiliary memory to main memory and, unless main memory is not full yet, some page in the main memory must be replaced using one of the replacement policies such as LRU, FIFO, or MRU.

The conventional storage hierarchy, consisting of main/auxiliary memory, was extended in the early 60's using an additional level called cache memory, which is a high-speed storage with a much faster access time than the main memory [Smith82]. Cache storage is extremely expensive compared to the main storage and therefore only small caches are typically used.

The address space is divided into equal blocks called pages and the main memory is divided into blocks of the same size called page frames. A page of data will reside in a page frame of memory, and the typical size of such a block is 512 to 1K words [Leung82]. Data transfers in the memory hierarchy are usually done by pages, rather than individual words or bytes, because locality of reference plays an important role in page transfer.

2.4 Cache Memory

Cache memory, as used in most computer systems, is a high-speed buffer memory interposed between main memory and the CPU. With the arrival of a logical address from the CPU, the operation of cache starts [Smith82]. At any time, cache contains most of the information that a processor needs. Whenever a reference is made to new data and that data is not present in cache, the old data in cache has to be replaced to give room to the

new data brought from main memory. So, in this context, the issues of data traffic between cache and main memory are analogous to the issues of data traffic between memory and auxiliary memory.

2.5 Cache Design Parameters

In uniprocessor computers, the main reason in employing a cache is to reduce the effective memory access time. If the miss ratio is reduced, the execution time can also be reduced. The execution time being "the sum of the time to service each cache hit plus the sum of the time to service each cache miss" [Marcovitz88]. If the misses occur close together (referred to as clumpiness of misses), then the time to service each cache miss can be less. Thus cache miss ratio can be a good performance metric in a single-processor, single-cache computer.

There are four important aspects to be considered in designing a cache memory [Smith82].

1) Improving the probability of finding a memory reference's target in the cache (the hit ratio).

2) Minimizing the miss ratio.

3) Minimizing the delay due to a miss.

4) Minimizing the overheads of updating main memory, i.e., whether to use a write-through or copy-back to reflect the modifications.

The following subsections describe the design parameters of a cache memory system such as cache size, block size, cache organization, misses in prefetch, misses

occurring in clumps, cache coherence, cache consistency, and replacement algorithms.

2.5.1 Cache Size

The size of the cache is an important design decision that impacts the performance and cost of a cache memory system. The larger the cache, the higher the probability of finding the required information in it [Smith82]. Obviously, cache cannot be expanded without limit, due to its cost and physical size.

2.5.2 Block Size

A block is a group of words that can be read from and written to a device. Selecting the block size is also an important decision that has to be considered in a memory system design. Kaplan and Winder [Kaplan73] indicated that there are a number of trade-offs in selecting the block size. Obviously, the transmission time for moving a small block from main memory to cache is less compared to that for a bigger block. Locality of reference plays an important role in making a decision about the block size. If the block size is large, the transmission time may be large, but the process can refer to the same block. If the block size is small, we may have to access main memory twice instead of just once. So the designer has to decide about the block size so as to improve the performance of the system.

2.5.3 Cache Organization

Cache organization is one of the design parameters that would influence the

performance and cost of a cache memory system. In order to locate an element in cache, it is necessary to have some kind of mapping which maps a main memory address to a cache location, or to search the cache associatively.

Various cache organizations such as fully associative, direct mapping, or set associative are used in most computer systems [Leung82]. The fully associative cache organization allows any page from main memory to be assigned to any page frame in cache. Figure 2 gives a clear picture of associative mapping. For each page of data stored, the corresponding main memory address is also stored. Whenever a reference is made, all the addresses are searched so as to find the match for the referenced address. In direct mapping cache organization, each page in memory can be mapped to a particular location in cache. This indicates that direct mapping is more restrictive than fully associative cache organization. Set associative cache organization involves organizing the cache into S sets of E elements per set. Thus the page frames in a set associative cache are grouped into a number of sets [Smith82]. Each page in main memory is mapped onto a page frame, which belongs to a particular set in cache. If a particular page is in cache, it must be stored in one of the elements in the corresponding set in cache. In this kind of cache organization, replacement policies will be made to the set of elements involved.

2.5.4 Misses in Prefetch

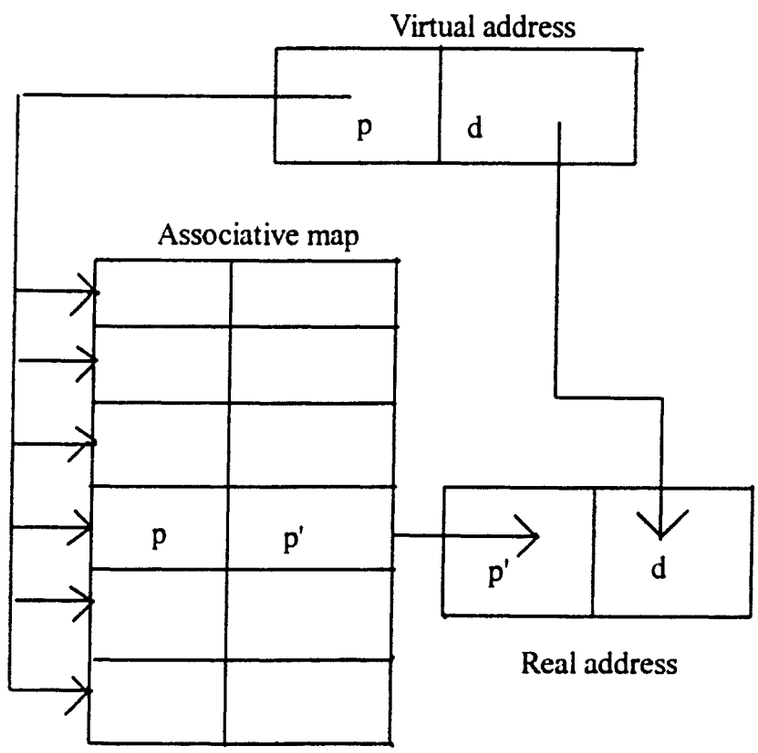
Prefetching is one of the popular strategies used to get the pages in cache before a particular page is required. Prefetching is used to get the data or instructions before they are actually needed by a program, with the intention that the program might use them in

the near future. In prefetching, the data that may be required in the near future is brought into the prefetch buffer.

There are two situations that can cause misses to occur when using prefetch buffers.

1) When the processor requests either data or instructions from main memory that is not available in cache, the processor has to wait till it gets the data; and

2) Network traffic to shared memory can increase the delay and can result in a cache miss.



p Virtual page number

p' Page frame number in main memory

d Displacement

Figure 2. Associative mapping using a page map table, given the virtual address

2.5.5 Misses Occurring in Clumps

Marcovitz discussed the clumpiness of misses, i.e., misses occurring close together for a shared memory multiprocessor with prefetching [Marcovitz88]. When misses are close together, the miss service times can be overlapped. When misses occur, it is good if they occur in clumps because the service time for those misses can be reduced. Hence the prefetch buffer has to wait for more than one miss to occur. Thus the number of misses that occur close together can be a good performance metric for a uniprocessor computer in designing a cache [Marcovitz88].

2.5.6 Cache Coherence

Cache coherence must be maintained when considering multiprocessor computers with shared memory and private caches. In these cases, the cache works like a uniprocessor's cache as long as a processor accesses data that is not shared with any other processor, keeping a copy of the recently used locations. In a uniprocessor environment, memory locations are shared only by a single processor, hence cache coherence need not be maintained as the processor can read the correct value. In a multiprocessor environment, the data in a particular location disappears from a processor's cache when another processor writes into it [Hill90]. When memory locations are shared among processors, cache coherence must be maintained so that each processor sees a correct value for the same variable. Marcovitz discusses cache coherence using non-cacheable marking [Marcovitz88]. Non-cacheable marking can help in maintaining cache coherence in a multiprocessor environment.

are typically used in order to replace the data in cache [Smith82]. The LRU policy using the stack model can be used to replace the information in cache. In the LRU stack model algorithm, the addresses referenced by the processor are placed in a stack with the most recently used address at the top of the stack and the least recently used address at the bottom of the stack. When a particular address is referenced, a search for the referenced block is carried out in the stack. The referenced address is then placed on the top of the stack and all other addresses are shifted down [Wang90].

CHAPTER III

DESIGN AND IMPLEMENTATION ISSUES

3.1 Implementation Platform and Environment

3.1.1 Sequent Symmetry S/81

The Symmetry S/81 is a powerful mainframe-class multiprocessor system developed by Sequent Computer System, Inc. Its shared-memory, multiprocessing architecture consists of the following elements [Sequent90]:

- A parallel architecture using multiple industry-standard microprocessors.
- The DYNIX/ptx or DYNIX V3.0 operating system, both UNIX system ports.
- Standard interfaces including Ethernet, MULTIBUS, VMEbus nad SCSI.

The operating system of the Symmetry S/81 have been engineered to incorporate parallel processing features. However, UNIX compatible software can run on the Symmetry S/81 without modification or with slight modification. In multi-user applications, tasks are automatically distributed to multiple processors which generally increases system throughput and reduces response times [Sequent90].

DYNIX V3.0 supports both the Berkeley UNIX and UNIX System V command sets, whereas DYNIX/ptx is compatible with AT&T System V3.2 only [Sequent90]. The simulation program for this thesis was developed on a Symmetry S/81 in C.

3.2 Objective

The main purpose of this thesis was performance analysis of cache using a trace-driven simulation technique. The simulation was run using address traces with variations in cache size, size of a page in cache, replacement algorithms, and cache access time. Simulation runs provided experimental results showing the performance changes (see Section 4.2) due to variations in those parameters.

3.3 Input Parameters

3.3.1 Trace Collection Method

Traces can be collected using certain UNIX utilities such as the ***prof*** command and *UilDumpSymbolTable* available on the Sequent Symmetry S/81 machine using DYNIX/ptx. These address traces serve as input to the simulation. The ***prof*** command is used in generating addresses referenced by programs during execution. The ***prof*** command interprets a profile file produced by the monitor function. Profiling is a three-step process. First a program is compiled with a -p option, then the program is executed, and finally the program is run to analyze the data. In DYNIX/ptx, the -p option to the C compiler command cc arranges for calls to monitor the addresses at the beginning and at the end of the execution and the profile file to be written [Sequent90].

Some of the traces used as input to the simulation were developed at the "Parallel Architecture Research Laboratory" of New Mexico State University [Spice94]. Gcc, spice, espresso, and eqntott were some of the traces that were developed on the dlx architecture

machine and are kept in the public directory of the ftp site tracebase@nmsu.edu [Spice94].

3.3.2 Cache Organization

A fully associative cache organization (see Section 2.5.3 for the definitions of various cache organizations) with page map tables and pages is used in this thesis to study the performance analysis of the cache. At any time, the cache contains page map tables and pages of the active jobs only. Several other cache organizations can also be used for performance analysis of cache.

3.3.3 Replacement policies

The LRU and FIFO replacement policies using a time-stamp are used in replacing the pages in cache in order to give room to new pages. The resident bit in the page map table plays an important role in the implementation of replacement policies.

3.3.4 Scheduling

A round-robin scheduling with time-slicing was used in this thesis work to simulate a multiprogramming environment. The choice of a particular scheduling algorithm can play an important role in improving the performance of a computer system.

3.4 Design of the Simulation

A trace-driven simulation has been developed on the Sequent Symmetry S/81

machine running the DYNIX/ptx operating system using the C programming language. The input to the simulation is a reference string of five jobs. The reference string of five jobs is stored in a reference file called REFILE. Each reference in the reference string contains two fields. The first field is the reference type and the second field is the memory address. Each reference in the reference file has a reference type and takes three values 0, 1, or 2. The value 0 or 1 indicates that a read operation needs to be performed, and the value 2 indicates that a write operation has to be performed.

An array of records has been used to simulate the cache. Once the user inputs the size of the cache, the array of records will be dynamically allocated according to the input value. Figure 3 gives a picture of the cache and main memory organization used in the simulation. A certain amount of the space in cache has been allotted for page map tables and a certain amount of space has been allotted for pages. At any time, the cache contains the page map tables and the pages of active jobs. The size of the page map tables is fixed and virtual memory is achieved through page traffic between the main memory and the cache. Figure 4 gives the data structure used in simulating the cache. Main memory contains the page map tables of all the jobs in the system. Whenever a job becomes active, a copy of the page map table is brought from the main memory and put in cache. Main memory also contains the global free frame table. This free frame table contains the information as to which page in the main memory is either allotted or available. Once a job terminates, all the page frames allotted to that job are made available for the other jobs through the free frame table. Figure 5 gives the data structure used in simulating the main memory.

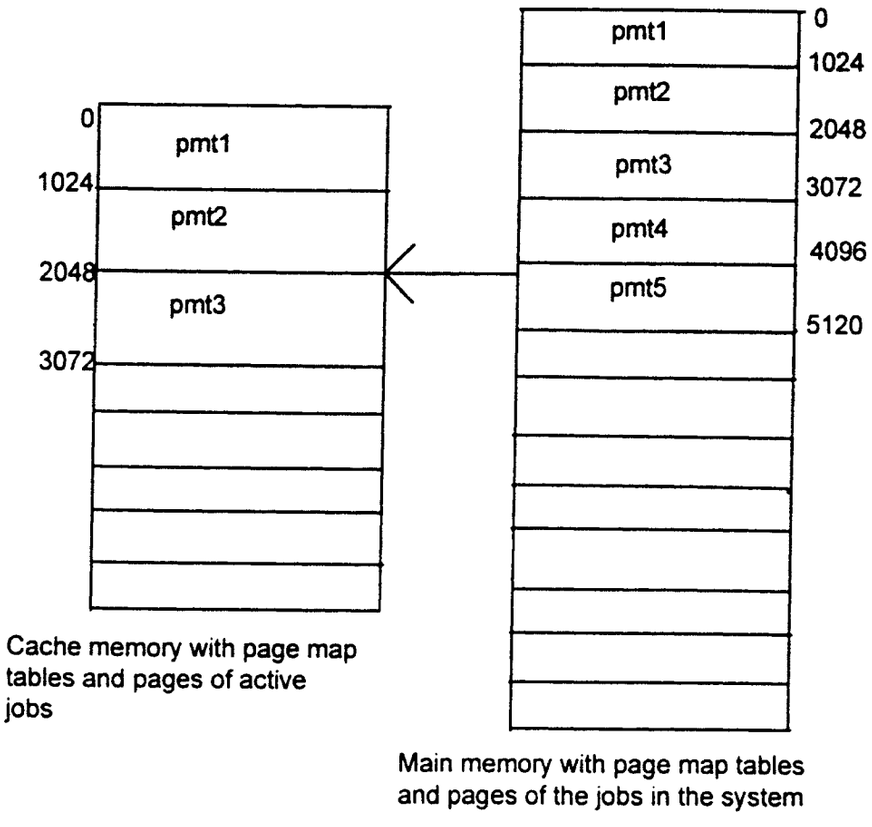


Figure 3. Organization of cache and main memory

3.4.1 Page Map Table

Page map table is used to map virtual addresses to physical addresses. A virtual address contains a virtual page number and an offset (a virtual address in general can contain a segment number also, but segmentation is beyond the scope of this thesis). The virtual page number is used as an index into the page table. From the page table entry, the page frame number is found. The page frame number is appended to the offset to form the physical address. The exact layout of each entry in the page map table is highly machine dependent, but the kind of information stored is almost the same from machine to machine. A typical page table entry has 32 bits, out of which 21 bits are allotted for

the frame number, 1 bit for the modified or dirty bit (to indicate if the referenced page has been modified), 1 bit used as the resident bit (to indicate if the page is in cache), and the remaining bits for caching. The resident bit plays an important role in several of the page replacement algorithms.

```
typedef struct {
        struct word pmpt[4096];
        struct page **pg;
        }CACHE;
```

Figure 4. Data structure of cache

```
struct mamem{
struct word pmt[5120];
struct page **mpg;
struct fft fmt[2000];
};
```

Figure 5. Data structure of main memory

3.4.2 Process Control Block

The PCB is a central store of information that allows the operating system to locate all key information about a process. When the operating system switches the CPU among processes, it uses the save areas in the PCB to hold the information such as the identification number of a process, the current state of the process, and the process' priority. Whenever a process gets the CPU, it uses the information stored in the PCB to

restart the process.

The pcb typically contains the following information for each job.

1) The job id.
2) When the job entered the system.
4) Number of pages allotted for the job.
5) The starting address of the job in cache.
3) The starting address of the page map table of the job in the main memory.

In this simulation program, a free PCB is obtained and allotted for a job whenever a job enters the system, and the job's identification number is stored in the PCB. Whenever the CPU switches among jobs, the jobs' current status is stored in the PCB so that, when the job gets back the CPU, the operating system can use the information stored in the PCB to restart the process.

3.5 Implementation Details

The main input to the simulation program is a reference string (also referred to as a trace) and the cache size. The reference file (REFILE) consists of reference strings for five jobs. Each reference in the reference file is processed separately. Some of the references used for this thesis are actual memory traces [Spice94]. To simulate a multi-user environment, the individual traces were interleaved.

The simulation program is menu driven. A user can input design parameters such as cache size and replacement policy, and obtain performance graphs generated by the system. The simulation has been implemented using the round-robin scheduling algorithm. In round-robin scheduling, a job is run till the time slice expires, the job terminates, the job asks for I/O, the job triggers a page fault, or the job asks for interprocess

communication, then the next job in the queue is given the CPU.

The cache used in this simulation contains the page map tables and pages of active jobs only. The size of the page map table and the number of pages allotted for each job are fixed and the maximum degree of multiprogramming is four. Thus, when four jobs are active, a copy of the four jobs' page map tables are brought into the cache from main memory. A fixed number of cache page frames are allotted to the active jobs when the cache is loaded. Each page of each job is mapped onto a distinct page in main memory via the page map table. Figure 6 gives the data structure used for the page map table in the simulation. The first page referenced by a job is always loaded into the cache and the resident bit for that page in the page map table is set to 1. The rest of the pages allotted for a job in cache are loaded upon request, using a demand page algorithm given in Figure 7. Each time a page is loaded into the cache, the resident bit for that particular page in the page map table is set to 1.

Once all the pages allotted for a job become full and a new page has to be brought in, one of the pages allotted for the job needs to be replaced using one of the replacement policies such as LRU or FIFO using the time-stamp. A variable called *clock* is used to indicate when a page was last referenced. A time-stamp is associated with each entry in the page map table and is used for implementing replacement policies. When a reference is made to a particular page and that page is not available in cache (i.e., a cache miss), the desired page has to be brought into the cache. The main memory page frame number is obtained from the page map table and the page is brought from the main memory and loaded into cache. Each time a page is referenced, the corresponding page map tables is

adjusted.

```
typedef struct word{
        int mapgno, residbit;
        char r_w;
        int time-stamp;
        int lpg_ca, modifibit;
        }WORDS;
```

Figure 6. The Data structure used for page map table

In the LRU (least recently used) policy, the entry in the page map table whose resident bit is set to 1 is checked to find the entry with the lowest time-stamp, and that page is replaced. In the FIFO (first in first out) policy, the entry in the page map table whose resident bit is set to 1 is checked to see which entry has the highest time-stamp value, and then that page is replaced. Figure 8 gives the pseudocode of the pagefault_handler algorithm used in the simulation. Once the job terminates, the cache is flushed and the main memory free frame table is adjusted accordingly.

The output of the simulation contains cache hit ratio, cache miss ratio, effective access time, and the delay caused due to cache miss. The delay due to a cache miss is calculated by the number of statements executed to get the page from main memory, and the effective access time is calculated by the sum of the time to service each cache hit plus the sum of the time to service each cache miss.

```
if(page_is_not_in_cache)
{
        if(nopages < nopagesalloted)
        {
                pagefault_handler();
        }
        else
        {
                obtain_mainframeno_from_pmt();
                load_cache();
        }
}
```

Figure 7. Demand page algorithm

```
find_the_least_recently_usedpage();
if(dirtybit_set)
{
        write_it_memory();
}
else
{
        replace_the_page();
}
```

Figure 8. Pagefault_handler algorithm

CHAPTER IV

EVALUATION OF THE SIMULATION

In this chapter, the evaluation of the simulation is mentioned with some observations based on the simulation. The results obtained through the simulation are compared against the results obtained by Marcovitz [Marcovitz88], Smith [Smith82], and Agarwal [Agarwal93].

4.1 Test Programs

Several traces obtained from the Parallel Architecture Research Laboratory of New Mexico State University were used to drive the simulation [Spice94]. The test programs that were used are gcc, spice, espresso, eqntott, and matrix. These traces were captured in real time from ten SPEC89 programs running on a Sun 3/60 under SunOS 4.0.3 [Spice94]. TABLE I gives the nature and characteristics of the traces used. Several graphical user interface application programs written in C were also used to collect address traces. These reference strings were also used to drive the simulation. The programs that were used were GNU chess, lander, xboard, xpaint, and CTWM. The miss ratios, hit ratios, delays due to a cache misses and execution times were obtained and the graphs were plotted to evaluate the simulation. A brief description of the programs, whose traces were obtained to drive the simulation, are given below.

TABLE I. TRACES USED FOR THE SIMULATION

Program	Length of the reference string
gcc	17, 432, 576
spice	22, 609, 920
espresso	13, 959, 168
eqntott	11, 599, 872
matrix	11, 592, 326
GNU chess	175
lunar lander game	142
xboard	132
xpaint	172
CTWM	95

Gcc Program: Gcc is the trace obtained from the GNU C compiler. The GNU C compiler is written in C. This benchmark "measures the time it takes for the GNU C compiler to convert a number of its pre-processed source files into optimized Sun-3 assembly language output" [Jhonson94].

Spice Program: Spice is the trace of the analog circuit simulator written in FORTRAN with a C interface to UNIX. This benchmark is a "general purpose circuit simulation program for nonlinear dc, nonlinear transient, and linear ac analyses" [Johnson94].

Espresso: Espresso trace is the trace obtained from a program used to minimize logic equations in computer design. This program is written in C.

Eqntott: Eqntott trace is the trace of a program that converts logic equations to truth tables.

Matrix: Matrix trace is a trace obtained from a matrix multiplication program written in C. This "benchmark also performs transposes using Linpack routines on matrices of order 300" [Johnson94].

GNU Chess: GNU chess is an ANSI/C version chess program developed by Stuart Cracraft.

Lunar Lander Game: The lunar lander game is a C implementation program of the old "lunar lander" game seen in amusement arcades. This program was developed using curses.

Xboard: Xboard is an X11/R4-based user interface for GNU chess.

Xpaint: Xpaint is also a graphical user interface program developed in X-windows. The program was developed by David Koblas used for drawing and editing figures similar to macpaint.

CTWM: CTWM (Claude's tab window manager) is a window manager for X-Windows.

4.2 GRAPHS

Graphs have been plotted using Harvard Graphics (Harvard Graphics for windows Ver 3.0, a software package developed by the Software Publishing Corporation), which is an interactive graphics package used to plot graphs. This tool was used to plot graphs with the hit ratio on the Y axis vs. the cache size on the X axis, or the miss ratio on the Y axis vs. the cache size on the X axis. Several graphs were plotted with different cache sizes and a fixed page size of 512 words per page, using the two different replacement policies of LRU and FIFO. Graphs were also plotted with the delay due to a miss on the X axis and the miss ratio on the Y axis for all the test programs. From the graphs obtained, it can be observed that the miss ratio can be a good performance metric in designing a cache. From the graphs, it can also be observed that the performance of a system can be improved by including page map tables and pages in cache, because the effective access time is less.

4.3 Observations

The graphs were plotted for all the test programs, and the graphs obtained were compared with the comparable graphs from the literature [Agarwal94] [Marcovitz88]. The graph in Figure 9 for the gcc trace (using the LRU policy) shows that as the cache size increases, the miss ratio decreases; but after a certain stage, the miss ratio is not affected even after increasing the cache size. The graph in Figure 10 for the gcc trace (using the LRU policy) shows that as the cache size increases hit ratio also increases. The graph in Figure 11, plotted for the gcc trace, shows that the delay due to a cache miss decreases

as the miss ratio decreases, because there are few misses and the amount of time to service a miss is less. The graph in Figure 12 shows that as the miss ratio decreases, the effective access time also decreases because the number of times the main memory is accessed to service a cache miss is less. The graphs (the delay due to cache miss vs. the miss ratio and the miss ratio vs. the effective access time) were compared with the graphs obtained by Marcovitz [Marcovitz88]. The graph in Figure 13 for the spice trace (with the LRU policy) does not show much difference in the miss ratio, even after increasing the cache size, mainly because of the reference pattern. The graph in Figure 17 for the espresso trace (with the LRU policy) shows that sometimes the miss ratio decreases and sometimes the miss ratio remains unchanged even after increasing the cache size, depending on the behavior of the program in execution. We can observe the same changes even in the hit ratio vs. the cache size. The graphs plotted were also compared with the results obtained by Agarwal [Agarwal93]. The graphs in Figures 14, 15, and 16 plotted for the spice trace, in Figures 18, and 19 plotted for the espresso trace, and in Figures 20, 21, and 22 plotted for the GNU chess trace can be analyzed in a similar way. So, by having the page map tables and the pages in cache, the effective access time is reduced. If effective access time is reduced, the overall execution time is also reduced and these results can be used in designing a cache.

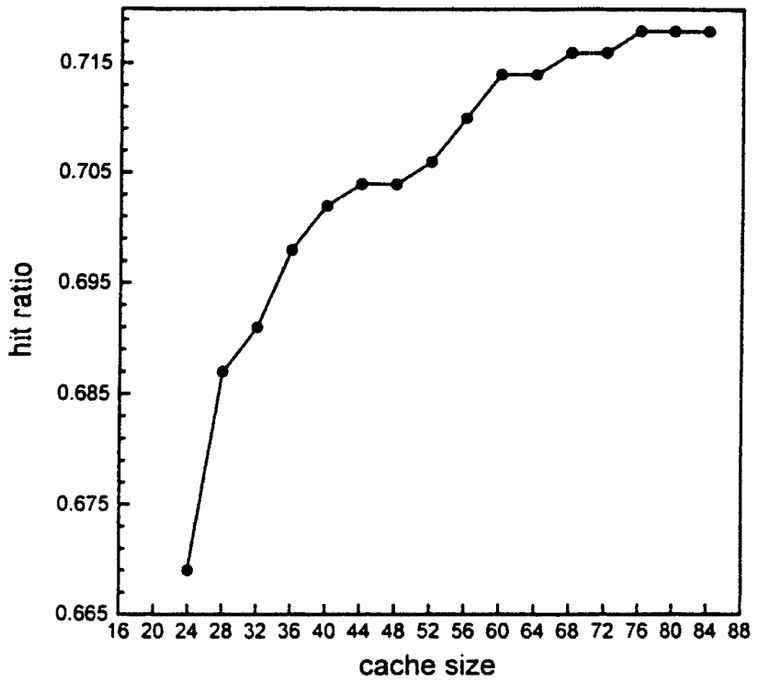


Figure 9. Hit ratio vs. cache size for gcc (LRU policy)

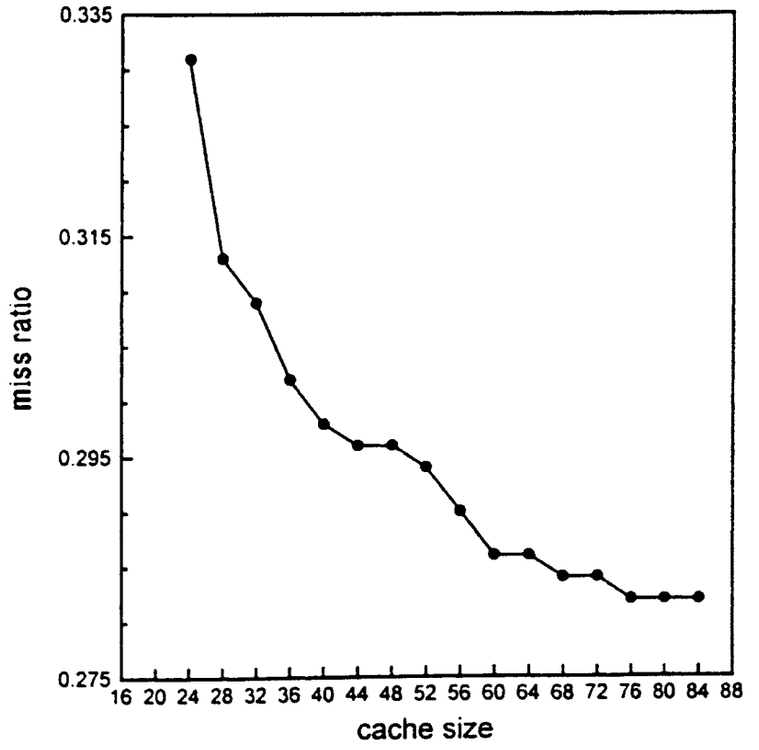


Figure 10. Miss ratio vs. cache size for gcc (LRU policy)

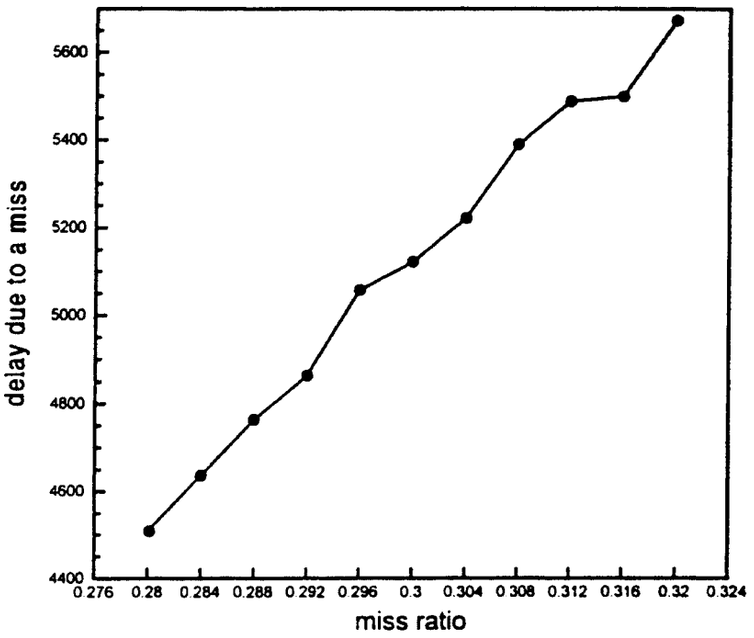


Figure 11. Miss ratio vs. delay due to a miss for gcc (LRU policy)

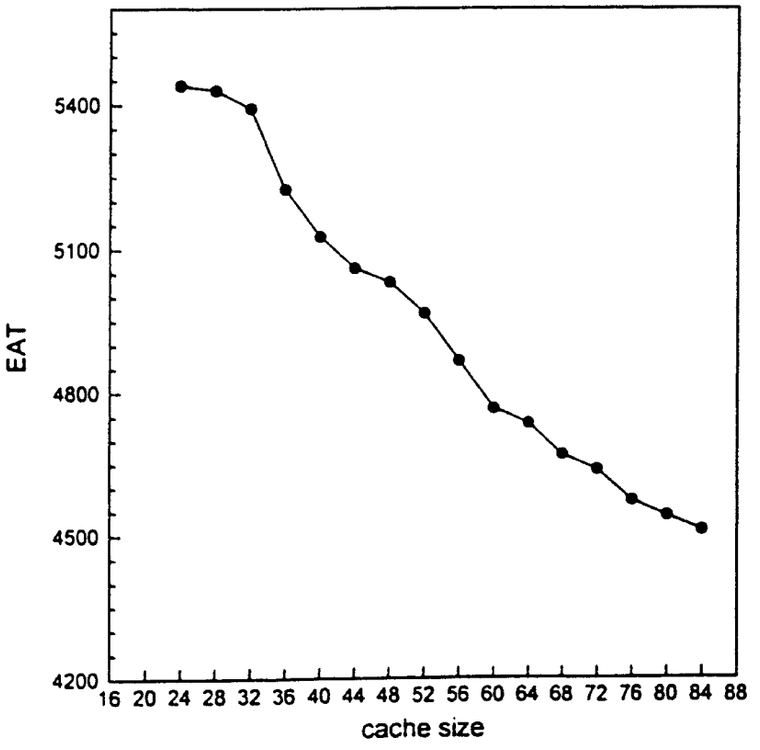


Figure 12. Cache size vs. effective access time for gcc (LRU policy)

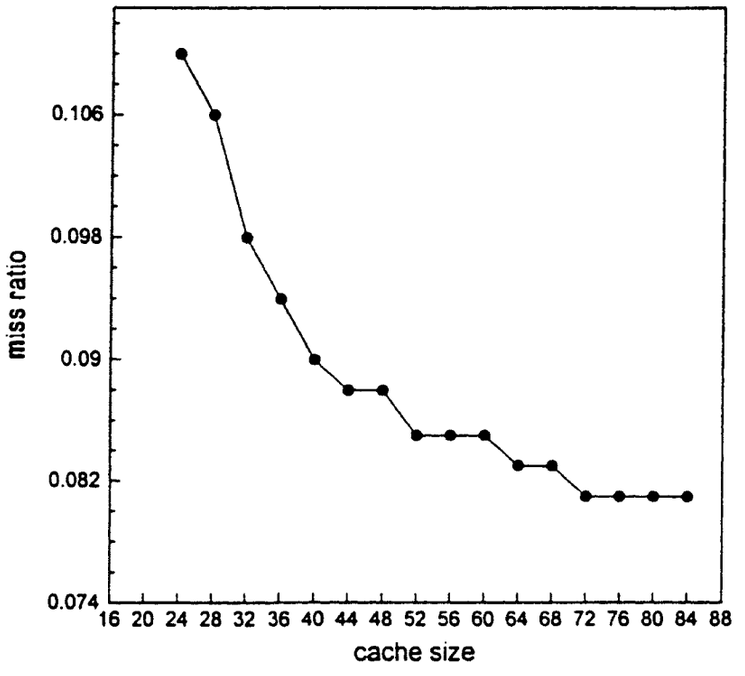


Figure 13. Miss ratio vs. cache size for spice (LRU policy)

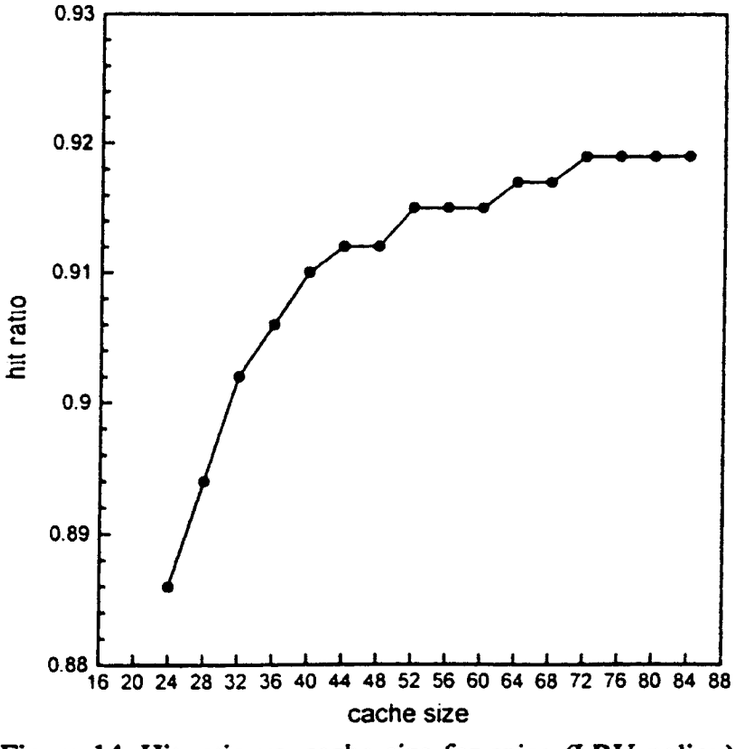


Figure 14. Hit ratio vs. cache size for spice (LRU policy)

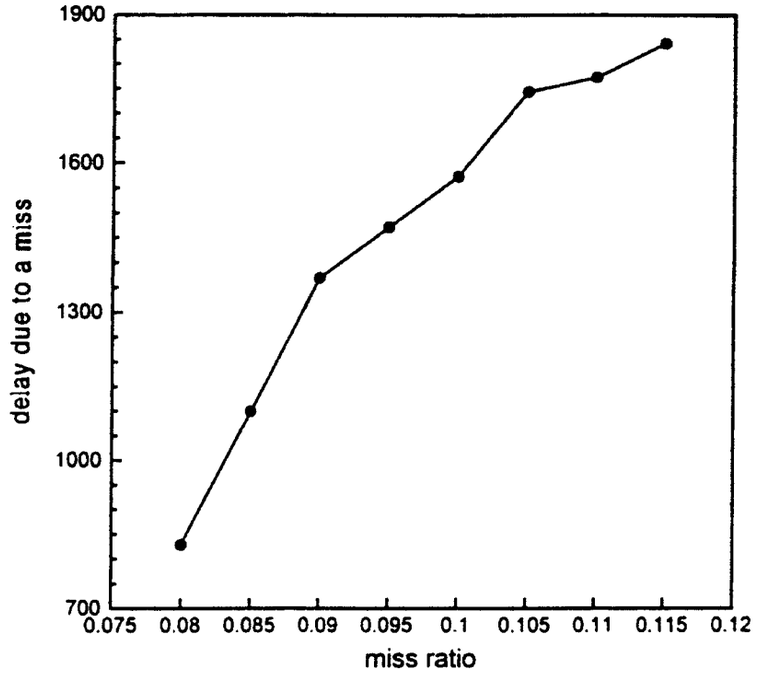


Figure 15. Miss ratio vs. delay due to a miss for spice (LRU policy)

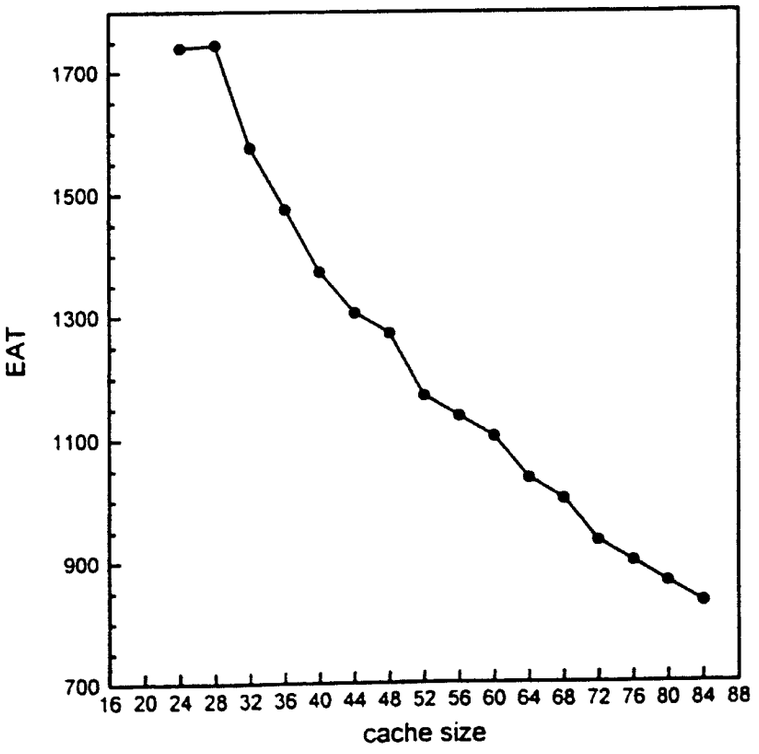


Figure 16. Cache size vs. effective access time for spice (LRU policy)

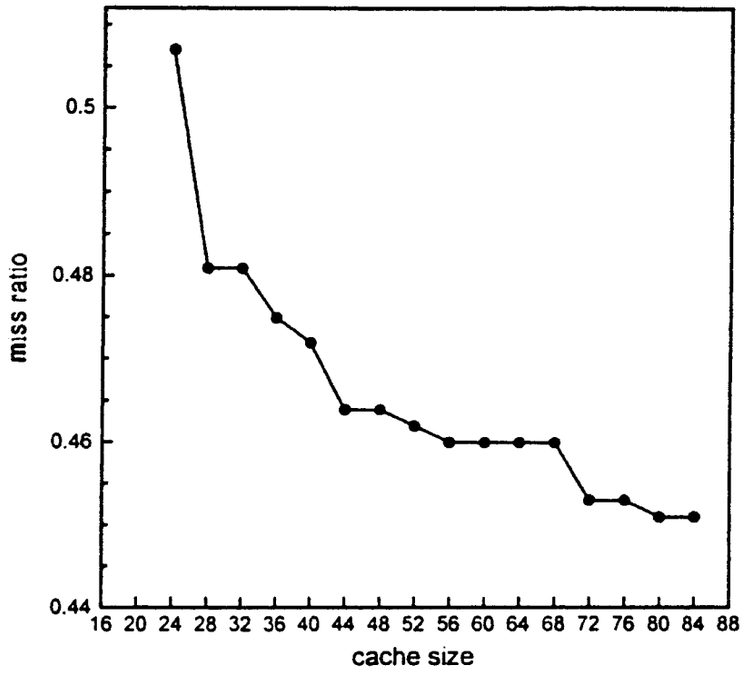


Figure 17. Miss ratio vs. cache size for espresso (LRU policy)

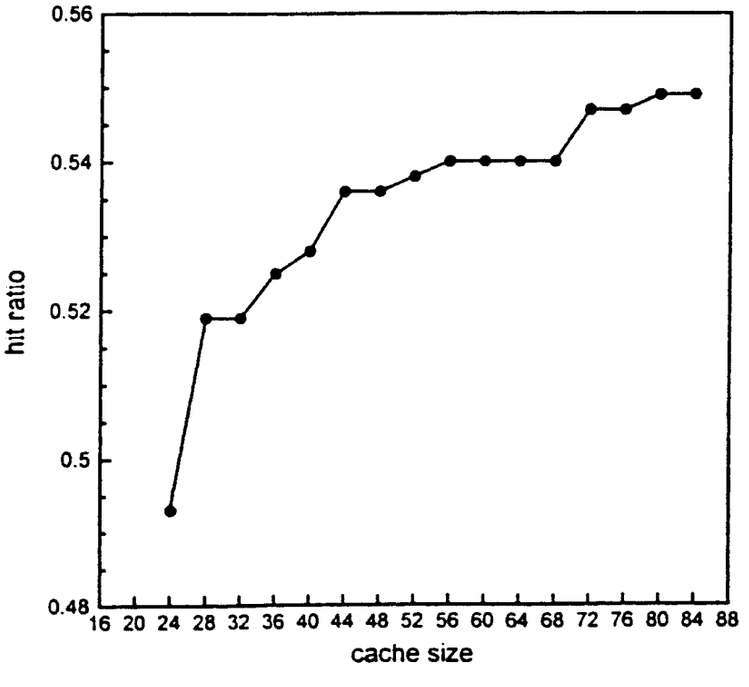


Figure 18. Hit ratio vs. cache size for espresso (LRU policy)

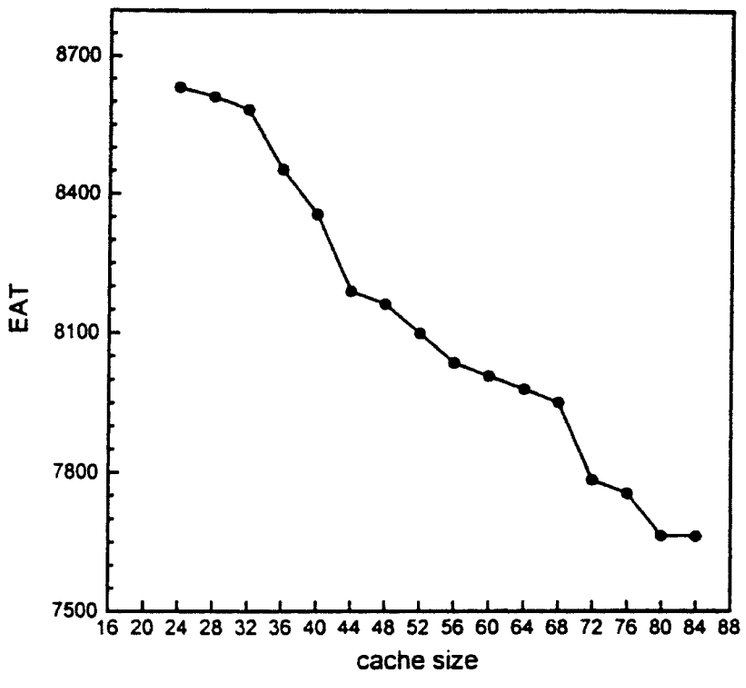


Figure 19. Cache size vs. effective access time for espresso (LRU policy)

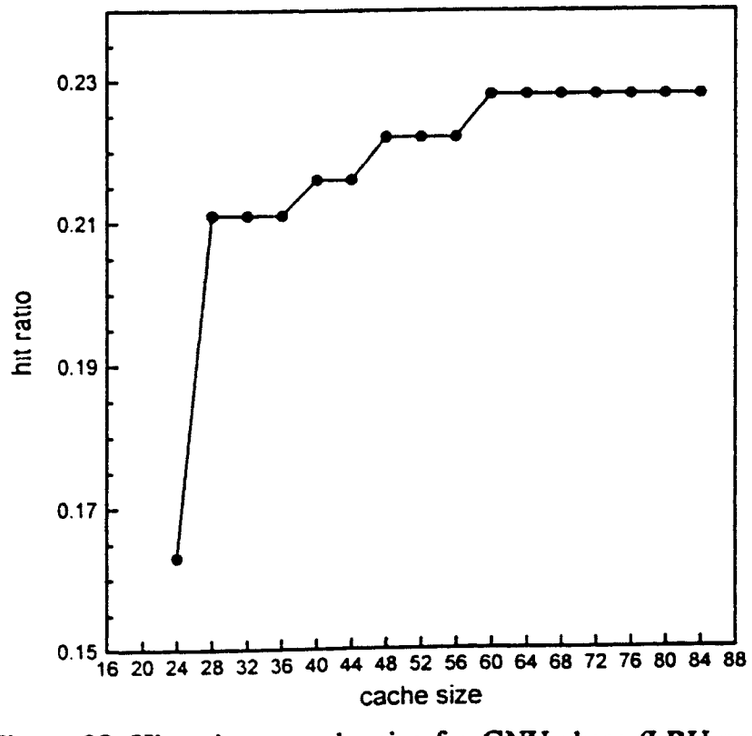


Figure 20. Hit ratio vs. cache size for GNU chess (LRU policy)

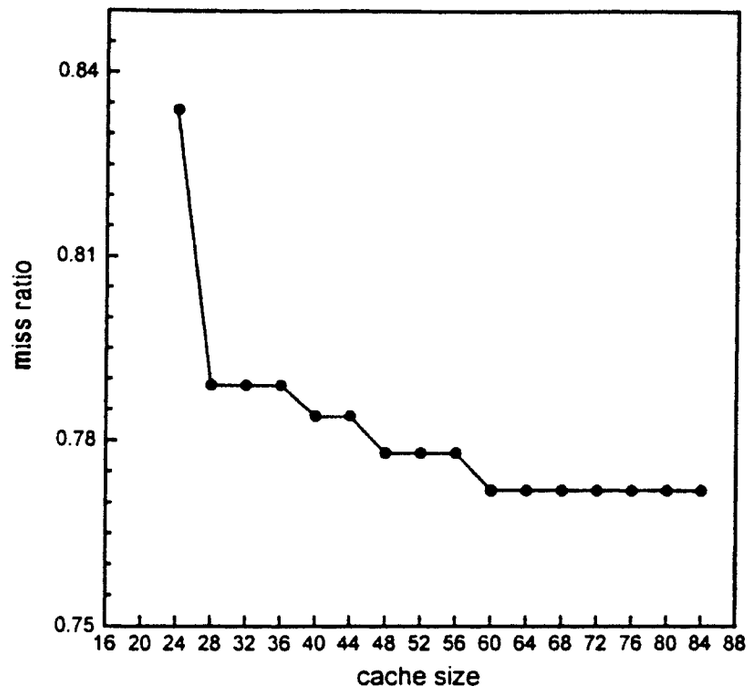


Figure 21. Miss ratio vs. cache size for GNU chess (LRU policy)

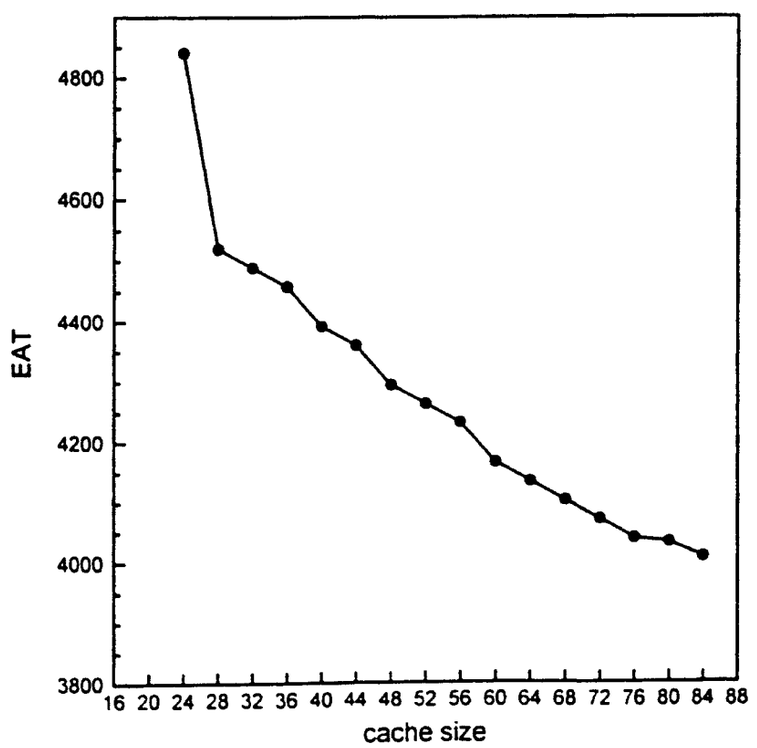


Figure 22. Cache size vs. effective access time for GNU chess (LRU policy)

CHAPTER V

SUMMARY AND FUTURE WORK

5.1 Summary

In Chapter I, the significance of the simulation, the introduction, and the main objective of the thesis was stated. Chapter II presented a general introduction to cache memory. The topics covered in this chapter consisted of the basic definitions to understand cache design, storage hierarchy, and some of the important cache design parameters such as cache size, block size, cache organization, replacement policy, cache coherence, snoopy cache mechanism, and cache consistency. Chapter III discussed the implementation issues and trace-driven simulation. Section 1 of Chapter III addressed the implementation platform and the run-time environment. Chapter III also contains the trace collection method, a brief description of page map tables, and other implementation details. Chapter IV discusses the evaluation of the simulation, the test programs used, and the graphs obtained.

The main objective of this thesis was to develop a simulation package for cache memory using a trace-driven simulation technique. This package can be used to design a system and improve the performance of an existing system. The results of this simulation were compared with the results obtained by Marcovitz [Marcovitz88], Agarwal

[Agarwal93], and Smith [Smith82].

5.2 Future Work

The future versions of this package should remove one or more restrictions mentioned below. The size of page map tables used in cache and main memory are fixed in the current implementation. The page map table size can be varied and allocated dynamically. A fixed number of page frames were allotted for each active job in cache. The number of page frames allotted for each job can be varied. Several other replacement algorithms such as second chance replacement, most recently used (MRU), and least frequently used algorithms can also be used as page replacement policies. Several other scheduling algorithms like FIFO (first in first out), SJF (shortest job first), and priority scheduling can also be used.

REFERENCES

[Agarwal88] Anant Agarwal, John Hennessey, and Mark Horowitz, "Cache Performance of Operating System and Multiprogramming Workloads", *ACM Transactions on Computer Systems*, Vol. 6, No. 4, pp. 393-431, November 1988.

[Agarwal93] Anant Agarwal and Steven D. Pudar," Column-Associative Caches: A Technique for Reducing the Miss Rate of Direct-Mapped Caches", *Proceedings of the 20th Annual International Symposium on Computer Architecture*, Los Alamitos, CA, USA pp. 179-190, 1993.

[Hill90] Mark D. Hill and James R. Larus, "Cache Considerations for Multiprocessor Programmers", *Communications of the ACM*, Vol. 33, No. 8, pp. 97-102, August 1990.

[Johnson89] Eric E. Johnson, "Working Set Prefetching for Cache Memories", *ACM Computer Architecture News*, Vol. 17, No. 6, pp. 37-141, December 1989.

[Johnson94] Collen S. Schieber and Eric E. Johnson, "RATCHET: Real-time Address Trace Compression Hardware for Extended Traces", *ACM Performance evaluation Reviews*, Vol. 21, No. 3, pp. 22-32, April 1994.

[Kaplan73] K. R. Kaplan and R. O. Winder, "Cache-Based Computer Systems", *IEEE Computer*, Vol. 6, No. 3, pp. 30-36, March 1973.

[Leung82] Yuk-Hoi Leung, "A Variable Cache Simulation System", *Project Report for Masters Degree*, University of Southwestern Louisiana, 57 pages, May 1982.

[Lilja93] David J. Lilja, "Cache Coherence in Large-Scale-Memory Multiprocessors: Issues and Comparisons", *ACM Computing Surveys*, Vol. 25, No. 3, pp. 303-338, September 1993.

[Lovett93] Tom Lovett, *Sequent Computer Systems, Inc.*, Personal Communication, June 1993.

[Marcovitz88] David Michael Marcovitz, "A Multiprocessor Cache Performance Metric", *Technical Report CSRD Rpt. No. 813 (UILU-ENG-88-8011), Centre*

for Supercomputing Research and Development, University of Illinois, Urbana, IL, August 1988.

[Sequent90] *DYNIX/ptx User's Guide*, Sequent Computer Systems, Inc., 1990.

[Smith82] Alan Jay Smith, "Cache Memories", *ACM Computing Surveys*, Vol. 14, No. 3, pp. 228-270, September 1982.

[Spice94] An International Trace Archive, *NMSU Tracebase*, New Mexico State University, Lascruses, NM, 1994.

[Stenstrom90] Per Stenstrom, " A Survey of Cache Coherence Schemes for Multiprocessors", *IEEE Computer*, Vol. 23, No. 6, pp. 12-24, June 1990.

[Stunkel91] Craig B. Stunkel, Bob Janssens, and W. Kent Fuchs, "Address Tracing for Parallel Mechanisms", *IEEE Computer*, Vol. 24, No. 1, pp. 31-38, January 1991.

[Wang90] Wen-Hann Wang and Jean Loup Baer, "Efficient Trace-Driven Simulation Methods for Cache Performance Analysis", *ACM SIGMETRICS: Performance Evaluation Review*, Vol. 18, No. 1, pp. 27-31, May 1990.

APPENDICES

APPENDIX A

GLOSSARY AND TRADEMARK INFORMATION

assembly level address tag:	One of the code insertion techniques in which a program is modified at the assembly level to generate addresses.
Cache Cherence:	Coherence is correctness; cache coherence means caches must be able to see the correct value for the same variable when a memory location is shared by different processors so as to maintain the correct execution of programs.
compiler based address tag:	One of the code insertion techniques in which a program is modified during compile time to generate addresses referenced by the processor.
gnuplot:	An interactive, command-driven function plotting program.
intermiss time:	The time between two misses on a single processor.
object level address tag:	One of the code insertion techniques (sometimes called as link time code modification) in which a program is modified during the link time for generating address traces.
pixie:	A program used to capture traces referenced by the processes during program execution.

TRADEMARK INFORMATION

APPENDIX B

PROGRAM LISTING

```
/*************************************************************************
  DESCRIPTION :
This program is used to study the performance of cache. A cache with
page map tables and pages has been used in the simulation. At any
instance, cache contains page map tables and pages of active jobs only.
Certain amount of storage in cache is reserved for page map tables and
certain amount of storage is used for pages. The jobs table gives the
information of the starting address of the job in cache. Each entry in
the page map table contains the page frame number, resident bit,
modified bit, location of the first page in cache, and time stamp. The
replacement policy used is LRU and FIFO to replace the page to give
room to the incoming page. The following information is obtained from
the  simulation. The cache miss ratio, the hit ratio, effective access
time, and the delay due to a miss.
*************************************************************************/

#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<string.h>
#include<malloc.h>

#define WORD 512                    /* size of the each page */
#define PMTSIZECA 8                 /* the storage space for page map table in
                                    cache*/
#define PMTSIZEMEM 10               /* the storage space for page map table in
                                    memory*/
#define PMTSIZE 1024                /* size of the page map table */
#define JOBSTOBEDONE 33             /* to check if the jobs are done */
#define MAXJOBS 200                 /* maximum number of jobs */
#define CACHEMISS 55                /* a global variable to check if it is a
                                    cache miss*/
#define LOADED 2                    /* a global variable to check if the
                                    page map table of the job is loaded
                                    successfully*/
#define CACHEFULL -1                /* a check to find if the cache is full */
#define ZZZ 99999                   /* variable used for random number
                                    generator*/
#define TERMINATED 99               /* a check to find if the job has
                                    terminated */
#define TIMEOVER 44                 /* check to see if the time has expired*/
#define TRUE 1                      /* a boolean variable for true or false*/
#define FALSE 0                     /* a boolean variable for true or false*/
#define AA 16807.0                  /* a variable used for random number
                                    generator*/
#define RR 2836.0                   /* a variable used for random number
```

```
                                generator*/
#define MM 2147483647.0         /* a variable used for random number
                                generator*/
#define QQ 127773.0             /* a variable used for random number
                                generator*/

/*****************************************************************************
The structure used for cache,main memory,page map table,pcb,list of jobs
in the system, list of jobs in the active queue.
*****************************************************************************/
/* structure of a pcb */
struct pcb_info{
      int jid;
      int jb_size;
      int start_time;
      int end_time;
      int pccounter;
      int base_addrcac;
      int base_addrmem;
      int lfirst_pgca;
      int nopages;
      int pcb_flg;
      int jb_status;
      };

/* structure declaration for the word */
/* each entry in page map table contains the following fields
      1. The main memory page frame number
      2. The resident bit to indicate whether the page is present
         in cache;
      3. The modified bit to indicate that the page has been modified
         since it has been last referenced.
      4. The time stamp used for the replacement policy.
      5. The location of the page in cache.
*/
typedef struct word{
      int mapageno;
      int residbit;
      int modifibit;
      int time_stamp;
      int lpg_ca;
      char oper;
      }WORDS;

/* structure declaration for the page*/
typedef struct page{
      struct word wrd[512];
      }PAGE;

/* structure declarations for cache */
typedef struct cache{
      struct page **pg;
      struct word pmpt[4096];
      }CACHE;

struct arr{
      int flag,no;
      int refno,index;
      };

typedef struct pmt{
      int base_pmtaddr,pmt_flag;
```

```
        int max;
        }PMT;

struct wrdbuf{
        int offst,ref_no;
        int termflg;
        };

/* structure for jobs table to give the no pages allotted */
struct jobs_table{
        int jbflg,list_id;
        int nopagesallt;
        };

/* structre for page list */
struct pg_list{
        int pgflg;
        };

struct temperory{
        struct wrdbuf wrdbf[1024];
        int buffjid;
        };

/* structure to collect information about all the jobs */
struct perf{
        int cache_hit;
        float cache_hitratio;
        float cache_missratio;
        int cache_miss;
        float hit_time;
        float miss_time;
        float update_time;
        float cupage_time;
        float page_time;
        int no;
        int job_id;
        };

/* structure for free frame table in main memory */
struct fft{
        int fftflg;
        };

/* structutre of main memory with free frame table and page map tables
*/
struct mamem{
        struct page **mpg;
        struct fft fmt[1000];
        struct word pmt[5120];
        };

int loading_cache();
int alljobs_loaded();
int obtain_pmtma();
int obtain_ind();
int obtain_pcb();
void adjust_jblist();
float get_random_no();
int get_job();
int run_job_timeslice();
int get_free_pgcache();
```

```
struct mamem *mem;
struct pcb_info pcb[MAXJOBS];
struct perf perf[MAXJOBS];
struct arr arry[1024];
struct pmt ca_pmtarry[10];
struct jobs_table list[MAXJOBS];
struct pg_list list_pges[1200];
struct temperory buffer[MAXJOBS];
struct pmt ma_pmtarry[MAXJOBS];

/* some of the global variables used */
int temp[25];
int cachesize = 0;
int numpages_cache = 0;
int perfcnt = 0;
double seed = 1.0;
int buffcnt = 0;
int clock_tick = 1;
int numpg_frames = 0,blksize = 0;
int pg_size = 0,pmtendaddr_ca = 0,pmtendaddr_ma = 0;
float delay = 0.0,hitser_time = 0.005;
CACHE *mycache;
FILE *fp1;
/*************************************************************************/
main()
{

      int choice,i = 0,j = 0;
      int time_slice = 0;
      int base_addrca = 0,base_addrma = 0,pmt_cntma = 0;
      int statcnt = 0,pmt_cntca = 0,numpmt = 0,config_no = 1;
      float exec_time = 0.0;
      int memsize = 0;
      char policy[6],scheduling[10];


      /* the menu used to drive the simulation */
      while (1)
       {
             system("tput clear");
          printf("\n\n\n\n\n\n\n\n\n");
          printf(" ****************************************\n");
          printf(" *              MENU                    *\n");
          printf(" *  ------------------------------      *\n");
          printf(" *                                      *\n");
          printf(" *  ENTER  0 -> ENTER THE CACHE SIZE    *\n");
          printf(" *  ENTER  1 -> PERFORMANCE ANALYSIS    *\n");
          printf(" *  ENTER  2 -> TO END THE SESSION      *\n");
          printf(" *                                      *\n");
          printf(" ****************************************\n");
          printf("\n            PLEASE ENTER NOW YOUR CHOICE -> ");
        scanf("%d%*c", &choice);
       system("tput clear");
       switch(choice)
           {
              case 0: config_no++;
                printf("\n CONFIGURATION NO : %d",config_no);
                clock_tick = 1;
                perfcnt = 0;
                buffcnt = 0;
                memsize = 512000;
                pg_size = 512;
```

```
time_slice = 0;
cachesize = 0;
base_addrma = 0;
base_addrca = 0;
blksize = 512;
numpages_cache = 0;
statcnt = 0;
numpg_frames = 0;
printf("\n ENTER THE CACHE SIZE NOW");
scanf("%d",&cachesize);

memset((struct pcb_info *)pcb,NULL,MAXJOBS *
sizeof(struct pcb_info));
memset((struct arr *)arry,NULL,1024 * sizeof(struct
arr));
memset((struct temperory *)buffer,NULL,MAXJOBS *
sizeof(struct temperory));

initialize_performance();
memset((struct pmt *)ma_pmtarry,NULL,MAXJOBS *
sizeof(struct pmt));
memset((struct pmt *)ca_pmtarry,NULL,10 *
sizeof(struct pmt));
memset((struct jobs_table *)list,NULL,MAXJOBS *
sizeof(struct jobs_table));
memset((struct pg_list *)list_pges,NULL,1200 *
sizeof(struct pg_list));

printf("\n ENTER THE REPLACEMENT POLICY AS LRU OR
FIFO");
scanf("%s",&policy);

strcpy(scheduling,"RR");
numpages_cache = (cachesize)/(pg_size) - PMTSIZECA;
cachesize = cachesize/512;

printf("\n CACHE SIZE IS                    :
%d",cachesize);
printf("K");
printf("\n NUMBER OF WORDS/PAGE IN CACHE   :
%d",blksize);
printf("\n NUMBER OF PAGES IN CACHE         :
%d",numpages_cache);
printf("\n REPLACEMENT POLICY               :
%s",policy);
printf("\n SCHEDULING                       :
%s",scheduling);
printf("\n");

/* main memory size is in bytes */
numpg_frames = ((memsize)/pg_size) - PMTSIZEMEM;

pmtendaddr_ca = (PMTSIZECA) * 512;
pmtendaddr_ma = (PMTSIZEMEM) * 512;
for(pmt_cntma = 0;pmt_cntma <=
(PMTSIZEMEM)/2;pmt_cntma++)
{
      ma_pmtarry[pmt_cntma].pmt_flag = 0;
      ma_pmtarry[pmt_cntma].base_pmtaddr =
      base_addrma;
      base_addrma = base_addrma + 1024;
}
```

```
numpmt = (PMTSIZECA / 2) - 1;
for(pmt_cntca = 0;pmt_cntca <= numpmt;pmt_cntca++)
{
      ca_pmtarry[pmt_cntca].pmt_flag = 0;
      ca_pmtarry[pmt_cntca].base_pmtaddr =
      base_addrca;
      base_addrca = base_addrca + 1024;
}

mycache = NULL;
mem = NULL;
/* allocating memory to cache dynamically */
mycache = (struct cache*)malloc(sizeof(struct cache));
if(mycache == NULL)
{
      printf("\n MEMORY ALLOCATION ERROR");
      exit(1);
}
/* allocating memory and initialising members in cache
*/
allocatemem_initialise_cache();
/* allocating memory for main memory */

mem = (struct mamem*)malloc(sizeof(struct mamem));
if(mem == NULL)
{
      printf("\n MEMORY ALLOCATION ERROR");
      exit(1);
}

allocatemain_initialise();

/* making page map tables for the jobs in the system
*/
memory_module();
for(i = 0;i <= buffcnt;i++)
{
      list[i].jbflg = 1;
      list[i].list_id = buffer[i].buffjid;
}
printf("\n JOBS STARTED EXECUTION");
if(strcmp(scheduling,"RR") == 0)
{
      printf("\n ENTER THE TIME SLICE");
      scanf("%d",&time_slice);
      printf("\n TIME SLICE : %d",time_slice);
      round_robin_scheduling(time_slice,policy);
}
else
      printf("\n ERROR IN TYPE OF SCHEDULING");

      break;
case 1: printf("\n PERFORMANCE STATISTICS");
statcnt = 1;
for(i = 0;i<= perfcnt;i++)
{
      exec_time = (float)(perf[i].cache_hit) +
      perf[i].page_time;
      printf("\n JOB ID : %d",perf[i].job_id);
      printf("\n CACHE HITS :
      %4.3f",(float)(perf[i].cache_hit)/(float)
      (perf[i].cache_hit + perf[i].cache_miss));
```

```
                        printf("\n CACHE MISS :
                        %4.3f",(float)(perf[i].cache_miss)/(float)
                        (perf[i].cache_hit + perf[i].cache_miss));
                        printf("\n DELAY TIME :
                        %4.3f",(perf[i].page_time));
                        printf("\n EXECUTION TIME : %4.3f",exec_time);
                        printf("\n UPDATE TIME :
                        %f",perf[i].update_time);
                        if((statcnt % 2) == 0)
                              getchar();
                        printf("\n\n");
                        statcnt++;
                  }
                  break;
               case 2: exit(1);
                        break;
               default : printf("\n\n\n\n\n\n                    INVALID
                  CHOICE. \n");
                          break;

            }
                  for(i = 0;i < (numpages_cache);i++)
                  {
                        free(mycache->pg[i]);
                  }
                  free(mycache->pg);
                  memset((struct word *)(mycache->pmpt),NULL,4096 *
                  sizeof(struct word));
                  free(mycache);
                  for(i = 0;i < numpg_frames;i++)
                  {
                        free(mem->mpg[i]);
                  }
                  free(mem->mpg);
                  memset((struct word *)(mem->pmt),NULL,5120 *
                  sizeof(struct word));
                  memset((struct fft *)(mem->fmt),NULL,1000 *
                  sizeof(struct fft));
                  free(mem);
                  fclose(fp1);
         printf("\n\n\n");
         printf("   PRESS <enter> TO CONTINUE =>   ");
         getchar();
         }
}
/*****************************************************************************
FUNCTION : initialize_performance()
PURPOSE  : This function is used to initialize the performance variables
*****************************************************************************/
initialize_performance()
{


      int i = 0;

      for(i = 0;i < MAXJOBS;i++)
      {

            perf[i].cache_hit = 0;
            perf[i].cache_hitratio = 0.0;
            perf[i].cache_missratio = 0.0;
            perf[i].cache_miss = 0;
```

```
            perf[i].hit_time = 0.0;
            perf[i].miss_time = 0.0;
            perf[i].update_time = 0.0;
            perf[i].cupage_time = 0.0;
            perf[i].page_time = 0.0;
            perf[i].no = 0;
            perf[i].job_id = 0;
      }



}
/*****************************************************************************
FUNCTION : allocatemem_initialise_cache()
PURPOSE  : This function is used to allocate memory dynamically to
           cache and initialise the cache
*****************************************************************************/
allocatemem_initialise_cache()
{


      int i = 0,k = 0;

      /* initialising page map tables */
      for(i = 0;i < pmtendaddr_ca;i++)
      {
            mycache->pmpt[i].mapageno = -1;
            mycache->pmpt[i].residbit = 0;
            mycache->pmpt[i].modifibit = 0;
            mycache->pmpt[i].time_stamp = -1;
            mycache->pmpt[i].oper = ' ';
            mycache->pmpt[i].lpg_ca = -1;
      }

      /*allocating memory dynamically to the page*/
      mycache->pg = (PAGE **)malloc(numpages_cache * sizeof(PAGE *));
      if(mycache->pg == NULL)
      {
            printf("\n MEMORY ALLOCATION ERROR");
            exit(1);
      }
      for(i = 0;i < numpages_cache;i++)
      {
            mycache->pg[i] = (PAGE *)malloc(sizeof(PAGE));
            if(mycache->pg[i] == NULL)
            {
                  printf("\n MEMORY ALLOCATION ERROR");
                  exit(1);
            }
            for(k = 0;k < pg_size;k++)
            {

                  mycache->pg[i]->wrd[k].mapageno = -1;
                  mycache->pg[i]->wrd[k].residbit = 0;
                  mycache->pg[i]->wrd[k].modifibit = 0;
                  mycache->pg[i]->wrd[k].time_stamp = -1;
                  mycache->pg[i]->wrd[k].oper = ' ';
                  mycache->pg[i]->wrd[k].lpg_ca = -1;
            }

      }
```

```
}
/*************************************************************************
 FUNCTION : allocatemain_initialise()
 PURPOSE  : This function is used to initialise the main memory
*************************************************************************/
allocatemain_initialise()
{


      int i = 0,k = 0;

      for(i = 0;i < pmtendaddr_ma;i++)
      {

            mem->pmt[i].mapageno = -1;
            mem->pmt[i].residbit = 0;
            mem->pmt[i].modifibit = 0;
            mem->pmt[i].time_stamp = -1;
            mem->pmt[i].oper = ' ';
            mem->pmt[i].lpg_ca = -1;

      }
      /*allocating memory dynamically to the page*/
      mem->mpg = (PAGE **)malloc((numpg_frames) * sizeof(PAGE *));
      if(mem->mpg == NULL)
      {
            printf("\n MEMORY ALLOCATION ERROR");
            exit(1);
      }

      for(i = 0;i < numpg_frames;i++)
      {
            mem->mpg[i] = (PAGE *)malloc(sizeof(PAGE));
            if(mem->mpg[i] == NULL)
            {
                  printf("\n MEMORY ALLOCATION ERROR");
                  exit(1);
            }

            /*allocate memory dynamically to the word*/
            for(k = 0;k < pg_size;k++)
            {
                  mem->mpg[i]->wrd[k].mapageno = -1;
                  mem->mpg[i]->wrd[k].residbit = 0;
                  mem->mpg[i]->wrd[k].modifibit = 0;
                  mem->mpg[i]->wrd[k].time_stamp = -1;
                  mem->mpg[i]->wrd[k].oper = ' ';
                  mem->mpg[i]->wrd[k].lpg_ca = -1;
            }
      }


}
/*************************************************************************
FUNCTION  : get_random_no()
PURPOSE   : This function returns a pseudo random number generator
            greater than or equal to zero and less than 1.The maximum
            int value taken is 32767
*************************************************************************/
float get_random_no()
{
```

```
        double hi,lo,test;

        hi = (int)(seed/QQ);
        lo = seed - QQ * hi;
        test = AA * lo - RR * hi;
        if(test > 0.0)
        {
              seed = test;
        }
        else
        {
              seed = test + MM;
        }
        return(seed/MM);

}
/*****************************************************************************
FUNCTION : random(n)
PURPOSE  : This function returns an integer between 0 and n-1.
*****************************************************************************/
random(n)
int n;
{

        double m;

        m = get_random_no();
        n = ((int)(m * 32767.0)) % n;

        return(n);
}
/*****************************************************************************
FUNCTION : memory_module()
PURPOSE  : This function is used to make the page map tables of the
           jobs and load the jobs into the system
*****************************************************************************/
memory_module()
{


        char S[80];
        int pmt_index = 0,pcb_index = 0;
        int flag,rnd = 0,lpgno = 0;
        char lopgno[5],type[5];
        int typ,count = 0,j_id = 0,cnt = 0;
        int max = 0,loca = 0,tempaddr = 0;

        /* opening the file for processing */
        fp1 = fopen("refstr","r");
        if(fp1 == NULL)
        {
              printf("\n ERROR OPENING INPUT FILE");
              exit(1);
        }

        /* obtain the page map table */
        pmt_index = obtain_pmtma();

        /* obtain a free pcb for the job  */
        pcb_index = obtain_pcb();

        fgets(S,80,fp1);
```

```
sscanf(S,"%s %s",&type,&lopgno);

/* storing all the references in a buffer */
if(strcmp(type,"JID") == 0)
{
      lpgno = atoi(lopgno);
      buffer[buffcnt].buffjid = lpgno;
}

/* mapping the logical addresses to the pages in main memory */
j_id = lpgno;
fgets(S,80,fp1);
sscanf(S,"%s %s",&type,&lopgno);
lpgno = atoi(lopgno);
typ = atoi(type);
max = 0;
/*numpg_frames = (numpg_frames - PMTSIZEMEM) + 1;*/
while(!(feof(fp1)))
{
      memset((struct arr *)arry,NULL,1024 * sizeof(struct arr));
      ma_pmtarry[pmt_index].pmt_flag = 1;
      pcb[pcb_index].pcb_flg = 1;

      /* storing the base address of the page map table in the
      pcb */
      pcb[pcb_index].base_addrmem =
      ma_pmtarry[pmt_index].base_pmtaddr;
      pcb[pcb_index].jid = j_id;
      perf[perfcnt].job_id = j_id;
      max = 0;
      tempaddr = pcb[pcb_index].base_addrmem;
      flag = TRUE;
      while((strcmp(type,"JID") != 0) && (flag == TRUE))
      {
            if(typ == 3)
            {
                  fgets(S,80,fp1);
                  sscanf(S,"%s %s",&type,&lopgno);
            }
            lpgno = atoi(lopgno);
            typ = atoi(type);
            buffer[buffcnt].wrdbf[cnt].ref_no = lpgno;
            cnt++;
            rnd = random(numpg_frames-1);
            while(mem->fmt[rnd].fftflg == 1)
            {
                  rnd = random(numpg_frames-1);
                  count++;
                  if(count >= 10000)
                  {
                        count = 0;
                        seed = seed + 1.0;
                        rnd = random(numpg_frames-1);
                  }
            }
            if(arry[lpgno].flag != 1)
            {
                  loca = tempaddr + lpgno;
                  mem->pmt[loca].mapageno = rnd;
                  arry[lpgno].flag = 1;
                  mem->fmt[rnd].fftflg = 1;
            }
```

```
                    else
                    {
                          loca = tempaddr + lpgno;
                    }
                    if((typ == 0) || (typ == 2))
                          mem->pmt[loca].oper = 'r';
                    else
                          mem->pmt[loca].oper = 'w';

                    /* making the flag of the occupied page to 1*/
                    if(lpgno > (max))
                    {
                          max = lpgno;
                          pcb[pcb_index].jb_size = max;
                    }
                    lpgno = 0;
                    fgets(S,80,fp1);
                    sscanf(S,"%s %s",&type,&lopgno);
                    flag = TRUE;
                    if(strcmp(type,"JID") == 0)
                    {
                          cnt = cnt - 1;
                          buffer[buffcnt].wrdbf[cnt].termflg = 1;
                    }
                    if(feof(fp1))
                    {
                          flag = FALSE;
                    }
              }
              if(flag == FALSE)
              {
                    cnt = cnt - 1;
                    buffer[buffcnt].wrdbf[cnt].termflg = 1;
                    break;
              }
              else
              {
                    lpgno = atoi(lopgno);
                    j_id = lpgno;
                    pmt_index = obtain_pmtma();
                    if(j_id == 1)
                          printf("\n STOP");
                    pcb_index = obtain_pcb();
                    buffcnt++;
                    buffer[buffcnt].buffjid = lpgno;
                    perfcnt++;
                    memset((struct arr *)arry,NULL,1024 * sizeof(struct
                    arr));
                    fgets(S,80,fp1);
                    sscanf(S,"%s %s",&type,&lopgno);
                    cnt = 0;
                    printf("\n\n");
              }
        }

}
/*************************************************************************
FUNCTION : loading_cache()
PURPOSE  : This function is used to load the page map table into cache
           and the page frames are allocated to the jobs depending on
           the size of the job.
```

```
*************************************************************************/
int loading_cache(id)
int id;
{
      int i = 0,prid = 0,loca = 0;
      int jd = 0,ref = 0,pcin = 0,addr_ma = 0;
      int main_no= 0,page_no,fr_pg = 0,num = 0;
      int locind = 0,locca = 0,sum = 0,locca_addr = 0;

      prid = id;
      for(i = 0;i < MAXJOBS;i++)
      {
            if(pcb[i].jid == prid)
                  break;
      }
      /* finding the pcb for the job given the job id*/
      pcin = i;

      for(i = 0;i <= buffcnt;i++)
      {
            if(buffer[i].buffjid == id)
                  break;
      }
      jd = i;
      /* address of the job in the main memory */
      addr_ma = pcb[pcin].base_addrmem;

      /* the available free pages in cache is obtained */
      fr_pg = get_free_pgcache();
      list_pges[fr_pg].pgflg = 1;
      pcb[pcin].lfirst_pgca = fr_pg;

      /* fixed number of page frames being allotted to each job */
      pcb[pcin].nopages = numpages_cache/4;
      num = pcb[pcin].nopages;
      num = num + fr_pg;

      /* setting the pages that have been allotted to each job as
      occupied*/
      for(i = fr_pg;i < num;i++)
      {
            list_pges[i].pgflg = 1;
      }
      locind = obtain_pmtca();

      /* if there is no free page map table available then
      the cache is returned full */
      if(locind == -1)
            return(CACHEFULL);
      else
      {
            /* if a free page map table is available then the page map
            table is loaded into cache */

            locca_addr = ca_pmtarry[locind].base_pmtaddr;
            pcb[pcin].base_addrcac = locca_addr;
            ca_pmtarry[locind].pmt_flag = 1;
            for(i = 0;i < 1024;i++)
            {
                  loca = addr_ma + i;
                  locca = locca_addr + i;
                  mycache->pmpt[locca].mapageno =
```

```
                    mem->pmt[loca].mapageno;
                    mycache->pmpt[locca].residbit =
                    mem->pmt[loca].residbit;
                    mycache->pmpt[locca].modifibit =
                    mem->pmt[loca].modifibit;
                    mycache->pmpt[locca].lpg_ca = mem->pmt[loca].lpg_ca;
                    mycache->pmpt[locca].time_stamp =
                    mem->pmt[loca].time_stamp;
                    mycache->pmpt[locca].oper = mem->pmt[loca].oper;
              }

              ref = buffer[jd].wrdbf[0].ref_no;
              main_no = mycache->pmpt[pcb[pcin].base_addrcac +
              ref].mapageno;
              mycache->pmpt[pcb[pcin].base_addrcac + ref].residbit = 1;
              mycache->pmpt[pcb[pcin].base_addrcac + ref].lpg_ca = fr_pg;
              mycache->pmpt[pcb[pcin].base_addrcac + ref].time_stamp =
              clock_tick;
              clock_tick++;
              for(i = 0;i < 512;i++)
              {

                    mycache->pg[fr_pg]->wrd[i].mapageno =
                    mem->mpg[main_no]->wrd[i].mapageno;
                    mycache->pg[fr_pg]->wrd[i].residbit =
                    mem->mpg[main_no]->wrd[i].residbit;
                    mycache->pg[fr_pg]->wrd[i].modifibit =
                    mem->mpg[main_no]->wrd[i].modifibit;
                    mycache->pg[fr_pg]->wrd[i].time_stamp =
                    mem->mpg[main_no]->wrd[i].time_stamp;
                    mycache->pg[fr_pg]->wrd[i].oper =
                    mem->mpg[main_no]->wrd[i].oper;
                    mycache->pg[fr_pg]->wrd[i].lpg_ca =
                    mem->mpg[main_no]->wrd[i].lpg_ca;

              }
              return(LOADED);
        }
}
/*****************************************************************************
FUNCTION : round_robin_scheduling()
PURPOSE  : This function is used to run the active jobs in a round
           robin fashion.
*****************************************************************************/
round_robin_scheduling(rrslice,repolicy)
int rrslice;
char repolicy[7];
{

        int cac_reply = 0,cpu_reply = 0;
        int actlcnt = 0,flg,actcnt = 0;
        int i = 0,jb_ind = 0,jb_id = 0,jld = 0;
        int j_id = 0,jobs_cnt = 0,NOMOREJOBS,numjobs = 0,ALLJOBSLOADED;
        int ALLJOBSDONE,deg_multi = 0;
        int pcbind = 0;
        struct jobs_table active_que[MAXJOBS];

        NOMOREJOBS = TRUE;
        numjobs = buffcnt;
        jb_ind = get_job(numjobs);

        /* get the first job in the system */
```

```
jb_id = list[jb_ind].list_id;

ALLJOBSLOADED = FALSE;
while(NOMOREJOBS == TRUE)
{
      /* loading the jobs until cache is full */
      cac_reply = loading_cache(jb_id);
      while((cac_reply != CACHEFULL) && (ALLJOBSLOADED != TRUE))
      {
            if(cac_reply == LOADED)
            {
                  deg_multi++;
                  active_que[actcnt].list_id = jb_id;
                  active_que[actcnt].jbflg = 1;
                  active_que[actcnt].nopagesallt++;
                  actcnt++;
                  list[jb_ind].jbflg = 0;
            }
            else
                  list[jb_ind].jbflg = 1;

            jld = alljobs_loaded();
            if(jld == -2)
                  ALLJOBSLOADED = TRUE;
            else
                  ALLJOBSLOADED = FALSE;

            jb_ind = get_job(numjobs);
            jb_id = list[jb_ind].list_id;
            cac_reply = loading_cache(jb_id);

      }
      acticnt = actcnt;
      j_id = active_que[0].list_id;
      active_que[actcnt].list_id = j_id;

      /* the job is run until time slice expires */
      cpu_reply =
      run_job_timeslice(rrslice,j_id,repolicy,active_que);
      active_que[actcnt].nopagesallt = active_que[0].nopagesallt;
      while(cpu_reply != TERMINATED)
      {
            for(i = 0;i < actcnt;i++)
            {
                  active_que[i].list_id = active_que[i+1].list_id;
                  active_que[i].nopagesallt =
                  active_que[i+1].nopagesallt;
            }
            active_que[actcnt].list_id = 0;
            active_que[actcnt].jbflg = 0;
            j_id = active_que[0].list_id;
            active_que[actcnt].list_id = j_id;
            cpu_reply =
            run_job_timeslice(rrslice,j_id,repolicy,active_que);
            active_que[actcnt].nopagesallt =
            active_que[0].nopagesallt;
      }
      printf("\n TERMINATED JID : %d ",j_id);
      active_que[0].list_id = 0;
      active_que[0].jbflg = 0;
      for(i = 1;i < actcnt;i++)
      {
```

```
                active_que[i-1].list_id = active_que[i].list_id;
            }
            actcnt--;
            pcbind = obtain_ind(j_id);
            numjobs--;
            adjust_jblist(numjobs,j_id);
            cache_flush(pcbind);
            jld = alljobs_loaded();
            if(jld == -2)
            {
                NOMOREJOBS = FALSE;
            }
            else
            {
                NOMOREJOBS = TRUE;
                jb_ind = get_job(numjobs);
                jb_id = list[jb_ind].list_id;
            }
        }

        /* till all jobs are done */
        ALLJOBSDONE = TRUE;
        while(ALLJOBSDONE == TRUE)
        {
            j_id = active_que[0].list_id;

            /* jobs being sent to CPU */
            cpu_reply =
            run_job_timeslice(rrslice,j_id,repolicy,active_que);
            if(cpu_reply == TERMINATED)
            {
                printf("\n TERMINATED JID : %d",j_id);
                active_que[0].list_id = 0;
                active_que[0].jbflg = 0;
                for(i = 1;i < actcnt;i++)
                {
                    active_que[i-1].list_id = active_que[i].list_id;
                }
                actcnt--;
            }
            else
            {
                /* if the job has not terminated, then the next job in
                the active queue is given the CPU */
                active_que[actcnt].list_id = j_id;
                for(i = 0;i < actcnt;i++)
                {
                    active_que[i].list_id = active_que[i+1].list_id;
                }
            }
            if(actcnt == 0)
                ALLJOBSDONE = FALSE;

        }
}
/*************************************************************************
FUNCTION : adjust_jblist()
PURPOSE  : This function is used to adjust the number of jobs in the
           system once the job terminates
*************************************************************************/
void adjust_jblist(njobs,njid)
int njobs,njid;
```

```
{
        int i = 0,j = 0;

        for(i = 0;i <= njobs;i++)
        {
                if(list[i].list_id == njid)
                        break;
        }
        list[i].jbflg = 0;
        for(j = i;j <= njobs;j++)
        {
                list[j].list_id = list[j+1].list_id;
        }
}
/*************************************************************************
FUNCTION : obtain_ind()
PURPOSE  : This function is used to obtain the correct job id when the
           active jobs page map table is to be loaded into cache
*************************************************************************/
int obtain_ind(id)
int id;
{

        int i = 0;

        for(i = 0;i < MAXJOBS;i++)
        {
                if(pcb[i].jid == id)
                        break;
        }

        return(i);
}
/*************************************************************************
FUNCTION : get_job()
PURPOSE  : This function is used to get the next job in the system
*************************************************************************/
int get_job(njbs)
int njbs;
{
        int i = 0;

        for(i = 0;i <= njbs ;i++)
        {
                if(list[i].jbflg == 1)
                {
                        break;
                }
        }
        if((i == njbs) && (list[i].jbflg == 0))
                return(-3);
        else
                return(i);
}
/*************************************************************************
FUNCTION : alljobs_loaded()
PURPOSE  : This function is used to check if all the jobs in the system
           are loaded and there are no more jobs in the system.
*************************************************************************/
int alljobs_loaded()
{
```

```
        int i = 0;

        /* finding out the number of jobs in the system */
        for(i = 0;i < MAXJOBS;i++)
        {
                if(list[i].jbflg == 1)
                {
                        break;
                }
        }
        if(i == MAXJOBS)
                return(-2);
        else
                return(0);
}
/*************************************************************************
FUNCTION : run_job_timeslice()
PURPOSE  : This function is used to run the jobs given the timeslice and
           the job is run till the time slice expires, Once the time
           slice expires and if the job has not terminated, the status
           of the job is kept in the program counter so the next time
           the job becomes active, the job can start its execution from
           the place where it has stopped.
*************************************************************************/
int run_job_timeslice(cpslice,rjid,rjpolicy,actlist)
int cpslice,rjid;
char rjpolicy[7];
struct jobs_table actlist[MAXJOBS];
{

        int main_no = 0,jbpages = 0,rnd_no = 0;
        int pgin_cache = 0;
        int pcb_id = 0,jb_run = 0,i = 0,j = 0,k = 0;
        int time = 0,prescnt = 0,ref = 0,main_frno = 0;
        int maddre = 0,pmtaddr = 0;
        int TERMFLG,check= 0,perfct = 0;

        for(i = 0;i < MAXJOBS;i++)
        {
                if(pcb[i].jid == rjid)
                        break;
        }
        pcb_id = i;
        jbpages = pcb[pcb_id].nopages;
        pgin_cache = pcb[pcb_id].lfirst_pgca;
        maddre = pcb[pcb_id].base_addrcac;
        for(i = 0;i <= buffcnt;i++)
        {
                if(buffer[i].buffjid == rjid)
                        break;
        }
        jb_run = i;
        jbpages = pgin_cache + jbpages;

        for(i = 0;i < MAXJOBS;i++)
        {
                if(perf[i].job_id == rjid)
                        break;
        }
        perf[i].no = perfcnt;
        perfct = i;
        time = 1;
```

```
prescnt = pcb[pcb_id].pccounter;
TERMFLG = 0;

/* running the job until time slice expires */
while(time <= cpslice)
{
      TERMFLG = 0;
      ref = buffer[jb_run].wrdbf[prescnt].ref_no;
      pmtaddr = maddre + ref;
      if(buffer[jb_run].wrdbf[prescnt].termflg == 1)
      {
            TERMFLG = 1;
      }
      if((TERMFLG == 0) || (TERMFLG == 1))
      {
            if(mycache->pmpt[pmtaddr].residbit == 1)
            {
                  perf[perfct].cache_hit++;
            }
            else
            {
                  check = actlist[0].nopagesallt;
                  j = pgin_cache + check;
                  if(j < jbpages)
                  {
                        delay = 0.0;
                        perf[perfct].cache_miss++;
                        mycache->pmpt[pmtaddr].residbit = 1;
                        delay++;
                        mycache->pmpt[pmtaddr].time_stamp =
                        clock_tick;
                        delay++;
                        mycache->pmpt[pmtaddr].lpg_ca = j;
                        delay++;
                        main_no = mycache->pmpt[pmtaddr].mapageno;
                        delay++;
                        for(k = 0;k < pg_size;k++)
                        {
                              mycache->pg[j]->wrd[k].mapageno =
                              mem->mpg[main_no]->wrd[k].mapageno;
                              mycache->pg[j]->wrd[k].residbit =
                              mem->mpg[main_no]->wrd[k].residbit;
                              mycache->pg[j]->wrd[k].modifibit =
                              mem->mpg[main_no]->wrd[k].modifibit;
                              mycache->pg[j]->wrd[k].time_stamp =
                              mem->mpg[main_no]->wrd[k].
                              time_stamp;
                              mycache->pg[j]->wrd[k].oper =
                              mem->mpg[main_no]->wrd[k].oper;
                              mycache->pg[j]->wrd[k].lpg_ca =
                              mem->mpg[main_no]->wrd[k].lpg_ca;

                        }
                        delay = delay + 6 * 512;
                        perf[perfct].page_time+= delay;
                        actlist[0].nopagesallt++;
                  }
                  else
                  {
                        delay = 0.0;
                        perf[perfct].cache_miss++;
                        main_frno =
```

```
                                    mycache->pmpt[pmtaddr].mapageno;
                                    delay++;
                                    if(strcmp(rjpolicy,"LRU") == 0)
                                    {

                                          pagefault_handler_LRUtime_stmp(
                                          main_frno, pcb_id, ref, perfct);
                                    }
                                    else if(strcmp(rjpolicy,"FIFO") == 0)
                                    {
                                          pagefault_handler_FIFOtime_stmp(
                                          main_frno,pcb_id,ref,perfct);
                                    }
                                    else
                                    {
                                          printf("\n ERROR IN CHOICE");
                                          exit(1);
                                    }
                              }
                        }
                  }
                  if(TERMFLG == 1)
                  {
                        return(TERMINATED);
                  }
                  if(time == cpslice)
                  {
                        pcb[pcb_id].pccounter = prescnt;
                        return(TIMEOVER);
                  }
                  prescnt++;
                  time++;
                  clock_tick++;
            }

}
/*************************************************************************
 FUNCTION : pagefault_handler_LRUtime_stmp()
 PURPOSE  : This function is used to replace the page in the cache to
            give room to the incoming page using a least recently used
            policy using time stamp.
*************************************************************************/
pagefault_handler_LRUtime_stmp(replno,perfjd,pgref,perfl)
int replno,perfjd,pgref,perfl;
{


      int i = 0,minimum = 0;
      int baddr = 0,j = 0,k = 0;
      int page_being_replaced = 0,rpgl = 0,rpg_frame = 0,pgno_cache = 0;
      int tempindex = 0;
      struct arr temp[30];


      /* Here an LRU replacement policy is used to replace the page
      using the time stamp */
      memset((struct arr *)temp,NULL,30 * sizeof(struct arr));
      delay++;
      baddr = pcb[perfjd].base_addrcac;
      delay++;
```

```
for(i = baddr;i <= (pcb[perfjd].jb_size + baddr);i++)
{
      delay++;
      if(mycache->pmpt[i].residbit == 1)
      {
            temp[j].no = mycache->pmpt[i].time_stamp;
            delay++;
            temp[j].refno = mycache->pmpt[i].mapageno;
            delay++;
            temp[j].index = i;
            delay++;
            j++;
            delay++;

      }
}

minimum = temp[0].no;
delay++;
page_being_replaced = temp[0].refno;
delay++;
tempindex = temp[0].index;
delay++;
for(k =     1;k < j;k++)
{
      delay++;
      if(temp[k].no < minimum)
      {
            minimum = temp[k].no;
            delay++;
            page_being_replaced = temp[k].refno;
            delay++;
            tempindex = temp[k].index;
            delay++;
      }
}

rpg1 = page_being_replaced;
delay++;
rpg_frame = tempindex;
delay++;
mycache->pmpt[rpg_frame].residbit = 0;
delay++;
mycache->pmpt[rpg_frame].time_stamp = -1;
delay++;
pgno_cache = mycache->pmpt[rpg_frame].lpg_ca;
delay++;
mycache->pmpt[rpg_frame].lpg_ca = -1;
delay++;


if(mycache->pmpt[rpg_frame].oper == 'w')
{
      for(i = 0;i < blksize;i++)
      {

            mem->mpg[rpg1]->wrd[i].mapageno =
            mycache->pg[pgno_cache]->wrd[i].mapageno;
            mem->mpg[rpg1]->wrd[i].residbit =
            mycache->pg[pgno_cache]->wrd[i].residbit;
            mem->mpg[rpg1]->wrd[i].modifibit =
            mycache->pg[pgno_cache]->wrd[i].modifibit;
```

```
                        mem->mpg[rpg1]->wrd[i].time_stamp =
                        mycache->pg[pgno_cache]->wrd[i].time_stamp;
                        mem->mpg[rpg1]->wrd[i].oper =
                        mycache->pg[pgno_cache]->wrd[i].oper;
                        mem->mpg[rpg1]->wrd[i].lpg_ca =
                        mycache->pg[pgno_cache]->wrd[i].lpg_ca;
                }
                perf[perfl].update_time+= 6 * (512 * 0.0005);
                delay++;
        }


        for(i = 0;i < blksize;i++)
        {

                mycache->pg[pgno_cache]->wrd[i].mapageno =
                mem->mpg[replno]->wrd[i].mapageno;
                mycache->pg[pgno_cache]->wrd[i].residbit =
                mem->mpg[replno]->wrd[i].residbit;
                mycache->pg[pgno_cache]->wrd[i].modifibit =
                mem->mpg[replno]->wrd[i].modifibit;
                mycache->pg[pgno_cache]->wrd[i].time_stamp =
                mem->mpg[replno]->wrd[i].time_stamp;
                mycache->pg[pgno_cache]->wrd[i].oper =
                mem->mpg[replno]->wrd[i].oper;
                mycache->pg[pgno_cache]->wrd[i].lpg_ca =
                mem->mpg[replno]->wrd[i].lpg_ca;
        }

        delay = delay + 6 * 512;
        /* here the page map tables are updated */
        mycache->pmpt[baddr + pgref].residbit = 1;
        delay++;
        mycache->pmpt[baddr + pgref].lpg_ca = pgno_cache;
        delay++;
        mycache->pmpt[baddr + pgref].time_stamp = clock_tick;
        delay++;
        perf[perfl].page_time+= delay;

}
/*****************************************************************************
FUNCTION : cache_flush()
PURPOSE  : This function is used to flush the cache once the job
           terminates
*****************************************************************************/
cache_flush(caind)
int caind;
{

        int i = 0,k = 0,m = 0;
        int addrca = 0,pmtind = 0,addrma = 0,loc = 0,pges = 0;
        int main_no = 0,totca = 0,pge = 0,totmem = 0;

        addrca = pcb[caind].base_addrcac;
        addrma = pcb[caind].base_addrmem;
        loc = pcb[caind].lfirst_pgca;
        pges = pcb[caind].nopages;
        totca = addrca + 1024;
        totmem = addrma + 1024;
        pmtind = obtain_pmtind(addrca);
        ca_pmtarry[pmtind].pmt_flag = 0;
        pcb[caind].pcb_flg = 0;
```

```
        pge = loc + pges;
        for(i = loc;i < pge;i++)
        {
              list_pges[i].pgflg = 0;
              for(k = 0;k < pg_size;k++)
              {
                    mycache->pg[i]->wrd[k].mapageno = -1;
                    mycache->pg[i]->wrd[k].residbit = 0;
                    mycache->pg[i]->wrd[k].modifibit = 0;
                    mycache->pg[i]->wrd[k].time_stamp = -1;
                    mycache->pg[i]->wrd[k].oper = ' ';
                    mycache->pg[i]->wrd[k].lpg_ca = -1;
              }
        }

        /* free frame table is set */
        for(i = addrca;i < totca;i++)
        {
              main_no = mycache->pmpt[i].mapageno;
              if(main_no != -1)
              {
                    for(m = 0;m < pg_size;m++)
                    {
                          mem->mpg[main_no]->wrd[m].mapageno = -1;
                          mem->mpg[main_no]->wrd[m].residbit = 0;
                          mem->mpg[main_no]->wrd[m].modifibit = 0;
                          mem->mpg[main_no]->wrd[m].time_stamp = -1;
                          mem->mpg[main_no]->wrd[m].oper = ' ';
                          mem->mpg[main_no]->wrd[m].lpg_ca = -1;
                    }
                    mem->fmt[main_no].fftflg = 0;
              }
              mycache->pmpt[i].mapageno = -1;
              mycache->pmpt[i].residbit = 0;
              mycache->pmpt[i].modifibit = 0;
              mycache->pmpt[i].time_stamp = 0;
              mycache->pmpt[i].oper = ' ';
              mycache->pmpt[i].lpg_ca = -1;
        }

        for(i = addrma;i < totmem;i++)
        {

              mem->pmt[i].mapageno = -1;
              mem->pmt[i].residbit = 0;
              mem->pmt[i].modifibit = 0;
              mem->pmt[i].time_stamp = 0;
              mem->pmt[i].oper = ' ';
              mem->pmt[i].lpg_ca = -1;
        }

}
/*****************************************************************************
 FUNCTION : obtain_pmtind()
 PURPOSE  : This function is used to get the page map table so as to
            flush the cache once the job terminates.
*****************************************************************************/
int obtain_pmtind(pmtaddrca)
int pmtaddrca;
{

      int i = 0,numpmts = 0;
```

```
        numpmts = PMTSIZECA / 2;
        for(i = 0;i < numpmts;i++)
        {
                if(ca_pmtarry[i].base_pmtaddr == pmtaddrca)
                        break;
        }

        return(i);
}
/*****************************************************************************
 FUNCTION : pagefault_handler_FIFOtimestamp()
 PURPOSE  : This function is used to replace the page in cache so as to
            give room to the incoming page using a first in first out
            replacement policy.
*****************************************************************************/
pagefault_handler_FIFOtime_stmp(fplno,fjid,fpgref,perff)
int fplno,fjid,fpgref,perff;
{

        int i = 0,maximum = 0;
        int j = 0,k = 0,baddr = 0;
        int replaced_page = 0,rpg1 = 0,rpg_frame = 0,pgno_cache = 0;
        int tempindex = 0;
        float upd_time = 0.0,pg_time = 0.0;
        struct arr temp[30];


        /* Here an LRU replacement policy is used to replace the page
        using the time stamp */
        memset((struct arr *)temp,NULL,30 * sizeof(struct arr));
        delay++;
        baddr = pcb[fjid].base_addrcac;
        delay++;

        for(i = baddr;i <= (pcb[fjid].jb_size + baddr);i++)
        {
                if(mycache->pmpt[i].residbit == 1)
                {
                        temp[j].no = mycache->pmpt[i].time_stamp;
                        delay++;
                        temp[j].refno = mycache->pmpt[i].mapageno;
                        delay++;
                        temp[j].index = i;
                        delay++;
                        j++;
                        delay++;
                }
                delay++;
        }

        maximum = temp[0].no;
        delay++;
        replaced_page = temp[0].refno;
        delay++;
        tempindex = temp[0].index;
        delay++;
        for(k =     1;k < j;k++)
        {
                if(temp[k].no > maximum)
                {
                        maximum = temp[k].no;
                        delay++;
```

```
                replaced_page = temp[k].refno;
                delay++;
                tempindex = temp[k].index;
                delay++;
        }
        delay++;
}

rpg1 = replaced_page;
delay++;
rpg_frame = tempindex;
delay++;
mycache->pmpt[rpg_frame].residbit = 0;
delay++;
mycache->pmpt[rpg_frame].time_stamp = -1;
delay++;
pgno_cache = mycache->pmpt[rpg_frame].lpg_ca;
delay++;
mycache->pmpt[rpg_frame].lpg_ca = -1;
delay++;


if(mycache->pmpt[rpg_frame].oper == 'w')
{
        for(i = 0;i < blksize;i++)
        {

                mem->mpg[rpg1]->wrd[i].mapageno =
                mycache->pg[pgno_cache]->wrd[i].mapageno;
                mem->mpg[rpg1]->wrd[i].residbit =
                mycache->pg[pgno_cache]->wrd[i].residbit;
                mem->mpg[rpg1]->wrd[i].modifibit =
                mycache->pg[pgno_cache]->wrd[i].modifibit;
                mem->mpg[rpg1]->wrd[i].time_stamp =
                mycache->pg[pgno_cache]->wrd[i].time_stamp;
                mem->mpg[rpg1]->wrd[i].oper =
                mycache->pg[pgno_cache]->wrd[i].oper;
                mem->mpg[rpg1]->wrd[i].lpg_ca =
                mycache->pg[pgno_cache]->wrd[i].lpg_ca;
        }
        perf[perff].update_time+= 6 * (512 * 0.0005);
        delay++;
}

for(i = 0;i < blksize;i++)
{

        mycache->pg[pgno_cache]->wrd[i].mapageno =
        mem->mpg[fplno]->wrd[i].mapageno;
        mycache->pg[pgno_cache]->wrd[i].residbit =
        mem->mpg[fplno]->wrd[i].residbit;
        mycache->pg[pgno_cache]->wrd[i].modifibit =
        mem->mpg[fplno]->wrd[i].modifibit;
        mycache->pg[pgno_cache]->wrd[i].time_stamp =
        mem->mpg[fplno]->wrd[i].time_stamp;
        mycache->pg[pgno_cache]->wrd[i].oper =
        mem->mpg[fplno]->wrd[i].oper;
        mycache->pg[pgno_cache]->wrd[i].lpg_ca =
        mem->mpg[fplno]->wrd[i].lpg_ca;
}
delay = delay + 6 * 512;
```

```
        /* here the page map tables are being updated */
        mycache->pmpt[baddr + fpgref].residbit = 1;
        delay++;
        mycache->pmpt[baddr + fpgref].lpg_ca = pgno_cache;
        delay++;
        mycache->pmpt[baddr + fpgref].time_stamp = clock_tick;
        delay++;
        perf[perff].page_time+= delay;
}
/*****************************************************************************
FUNCTION : obtain_pmtma()
PURPOSE  : This function is used to get the page map table in the memory
           so as to load the job into the system
*****************************************************************************/
int obtain_pmtma()
{
        int i = 0;

        for(i = 0;i < 20;i++)
        {
                if(ma_pmtarry[i].pmt_flag != 1)
                        break;
        }
        return(i);

}
/*****************************************************************************
FUNCTION : obtain_pmtca()
PURPOSE  : This function is used to get the page map table in cace so as
           to load the active jobs page map table in cache
*****************************************************************************/
obtain_pmtca()
{

        int i = 0;
        int numpmts = 0;

        numpmts = PMTSIZECA/2;
        for(i = 0;i < numpmts;i++)
        {
                if(ca_pmtarry[i].pmt_flag != 1)
                {
                        break;
                }
        }
        if(i == numpmts)
                return(-1);
        else
                return(i);

}
/*****************************************************************************
FUNCTION  : obtain_pcb()
PURPOSE   : This function is used to get the pcb for the job once the
            job enters the system.
*****************************************************************************/
int obtain_pcb()
{

        int i = 0;

        for(i = 0;i < MAXJOBS;i++)
```

```
        {
                if(pcb[i].pcb_flg != 1)
                        return(i);
        }
}
/*****************************************************************************
FUNCTION: get_free_pgcache()
PURPOSE : This function is used to get the free page in cache to allot
          for the job once the job becomes active
*****************************************************************************/
int get_free_pgcache()
{
        int i = 0;

        for(i = 0;i < numpages_cache;i++)
        {
                if(list_pges[i].pgflg != 1)
                        break;
        }
        return(i);
}
/*****************************************************************************/
```

VITA

Pamela Neelaveni

Candidate for the degree of

Master of Science

Thesis: CACHE PERFORMANCE ANALYSIS: A TRACE-DRIVEN SIMULATION

Major Field: Computer Science

Biographical:

Personal Data: Born in Hyderabad, INDIA, on December 18, 1968, daughter of N. Sreeram and N. Chandra Leela.

Education: Graduated from St. Anns Junior College, Hyderabad, INDIA in May 1985; received Bachelor of Engineering (Hons) degree in Chemical Engineering from Birla Institute of Technology and Science, Pilani, Rajasthan, INDIA in June 1990. Completed the requirements for the Master of Science degree in Computer Science at the Computer Science Department at Oklahoma State University in July 1994.

Experience: Worked as design engineer for Gwalior Rayon Industries; employed by Oklahoma State University, University Computer Center as a graduate research assistant from October 1992 to June 1994.