Dynamically Adaptive Prepaging for Effective Virtual Memory Management

Dean Morrison
Dept. of Electrical and Computer Engineering
University of Florida
dean8201@ufl.edu

12/06/2006
EEL 6892 – Virtual Computers
Acting Supervisor, Dr. Renato J. Figueiredo

Abstract

Demand prepaging, an extension to the widely employed method of demand paging, is a concept designed to reduce page faults in a system implementing virtual memory by prefetching pages speculated to be referenced in the near future in addition to pages that have already been referenced. Doing so exploits high disk bandwidths while attempting to avoid high disk latencies.

Although studies have shown that demand prepaging is generally not beneficial, other more recent studies have been able to demonstrate significant page fault reductions using dynamic prepage parameters as opposed to static [4]. There are several major parameters associated with demand prepaging however, and to date there are no studies which evaluate the performance of a completely dynamic set of these parameters.

In this paper, I propose and evaluate a Dynamically Adaptive Prepaging (DAP) scheme in which all major demand prepaging parameters are dynamically modulated to changes in reference stream trends and phases. My proposed DAP policies attempt to reduce page faults by exploiting high disk bandwidths.

I evaluate my proposed system through simulation, using a custom trace-driven simulator developed in C++. Various uni-programmed memory reference traces were simulated. My evaluation shows a potentially significant reduction in page faults for the simulated traces.

1. Introduction

With processor and main memory speeds increasing ever more rapidly, the gap between storage disks and memory is growing wider every day. As this gap grows larger, so does the latency associated with the handling of page faults – an event in which a virtual memory reference translates to a physical page not currently located in physical memory. This increasing latency is why reducing virtual memory page faults is a growing interest.

Demand paging, a concept as old as virtual memory itself, is a technique for implementing virtual memory in which only demanded pages are fetched from the backing store into physical memory. An extension to this technique is the concept of *demand prepaging* which was proposed as a means for reducing page faults. Just as instruction prefetching is used in processor pipelines to maximize pipeline utilization, so is demand prepaging used to reduce disk accesses caused by page faults. With demand prepaging, the O/S (or VMM) is permitted to fetch extra pages in addition to the demanded page. If the prepaged pages are referenced relatively soon after being fetched, the demand prepaging system successfully averted additional disk accesses.

The effectiveness of demand prepaging relies on the fact that the bulk of the delay associated with retrieving a page from disk storage is due to the latency of the disk access (typically 1 to 10 ms). The transfer delay – the time required to transmit the page from the backing store into physical memory – is significantly smaller (on the order of tenths of a millisecond).

Therefore, the additional delay caused by "piggy-backing" the transfer of extra pages along with the demanded page can be considered negligible. Since disk bandwidths have been improving at a greater rate (approximately 20% per year) than disk latencies (approximately 5% per year), it is clear that reducing the number of page faults is more important than reducing the number of pages transferred between disk and main memory.

1.1 Demand Prepaging

As previously mentioned, demand prepaging is an extension to demand paging that allows extra pages to be fetched into main memory along with the demanded page. There are several parameters associated with demand prepaging, the most important of which (and those that will be analyzed in this paper) are:

- *Prepage Memory Allocation*: The number of main memory page frames to be allocated for prepaged pages – pages that are fetched along with the page whose reference cause a page fault.
- P*repage Degree:* The number of additional pages to fetch along with the page whose reference caused a page fault.
- *Prepage Prediction Method*: The method used to predict pages that are most likely to be referenced in the near future and should therefore be candidates for prepaging.

It is intuitive that there exists an optimum configuration of these parameters that will minimize page faults.

The concern with applying prepaging techniques to virtual memory management, however, is the (somewhat likely) possibility of causing more page faults than would occur in a system that does not use prepaging. This, along with obvious drawbacks of increased kernel complexity and strain on disk bandwidth, are factors which must be overcome in order to benefit from the application of prepaging.

Many prepaging techniques have been suggested, from the simple *One Block Lookahead (OBL)* policy [1] to the somewhat more complex *OBL/k* policy proposed by Horspool and Huberman [3]. Until recently, the problem with most of the suggested prepaging techniques is simply that they fail to substantially reduce page faults over a broad range of workloads. Kaplan et al. [4] propose a means to dynamically adapt prepaging policies in order to accommodate various workloads. This was done by dynamically adjusting prepage memory allocation, or what they refer to as *target allocation* or *prepaged allocation*.

1.2 Dynamically Adaptive Prepaging (DAP)

Although dynamically adjusting prepaged allocation will surely help the O/S (or VMM) adapt to changes in memory reference behavior, it fails to fully exploit the mutual relationship between all of the prepaged parameters. This is why I propose to design and evaluate more advanced methods for dynamically adapting not only prepage memory allocation, but prepage degree and prediction as well. It is reasonable to believe there should exist a combination of these three parameters that optimizes the performance (minimizes page faults) of a virtual memory management system. The challenge lies in developing methods to adjust these parameters in order to adapt to changing memory reference behavior without incurring an unacceptable overhead in the process.

2. Background and Related Work

It is important to understand the consequences of prepaging and the adjustment of its parameters on the efficacy of virtual memory management before trying to optimize it. The following section will discuss the various costs and benefits of different prepaging strategies.

2.1 Prepaging: Costs and Benefits

In this paper, I will refer to pages that are resident in main memory as a result of being referenced by the memory management unit as *used* pages. Pages that are resident but have not yet been referenced are called *prepaged* pages.

Prepaging is beneficial only if pages that are prepaged are referenced before being evicted. If evicted having never been referenced, the prepaged page only occupied a main memory page frame which could have been better used by a used page. This means that prepaging is harmful only if prepaged pages displace used pages that, under a non-prepaging system, would have been re-referenced before being evicted. So, one could characterize the *cost* of prepaging as the number of references to pages that are not resident under the given prepaging policy but would have been resident had prepaging not been used. Then, the *benefit* of prepaging is the number of references to prepaged pages that would not have been resident without prepaging.

2.1.1 Prepage Allocation

As previously mentioned, *prepage allocation* is the number of main memory page frames to allocate to prepaged pages. The allocation of main memory page frames between used and prepaged pages is a source of contention that will be managed by the O/S (or VMM).

The two extremes of prepage allocation lead to associated costs and benefits. If prepage allocation is too high, less memory is allocated to used pages which could result in an increase in page faults. If it is too low, this reduces the effectiveness of prepaging altogether.

2.1.2 Prepage Degree

Prepage degree is the number of additional pages to fetch along with the demanded page when handling a page fault. If the degree is too low, the effectiveness of prepaged page prediction is reduced. If the degree is too high, however, previously prepaged pages may be evicted before ever being referenced in order to make room for the large amount of new prepaged pages. Recall that prepaging is beneficial only if pages that are prepaged are referenced before being evicted. Therefore, a high prepage degree may reduce the potential for prepaging to be beneficial.

2.2 Previous Research in Prepaging

Many papers evaluate the benefits of demand prepaging. It is important to review their findings before attempting to develop new, more effective methods.

2.2.1 One Block Lookahead (OBL)

Perhaps the earliest and most minimal prepaging concept was Joseph's *One Block Lookahead (OBL)* policy [1]. Under this policy, if a reference to page *p* causes a page fault, page *p*+1 will be fetched along with page *p* if it is not already resident in main memory. The main purpose of such a policy was to exploit spatial locality in memory pages.

Joseph also suggested loading prepaged pages into the LRU position of the resident page queue, as a way to avoid evicting used pages before prepaged pages. In doing so, however, the benefits of prepaging were minimized since a prepaged page must be used prior to the next page fault in order not to be evicted. It is no surprise that experiments with OBL yielded little to no benefit.

Smith [2] modified OBL such that pages *p* and *p*+1 are placed in the first and second positions of the LRU queue. This essentially amounts to a prepaging policy with a prepaged allocation of 50%. This relatively high prepaged allocation is most likely why this policy *increased* page faults more often than not.

2.2.2 OBL/k

Horspool and Huberman [3] proposed a more complex extension to OBL in which prepaged pages advanced towards eviction at a rate *k*. This bias toward preserving used pages proved effective as they were able to demonstrate a modest reduction in page faults.

2.2.3 Adaptive Caching

Kaplan et al. [4] proposed a means to dynamically adapt prepaging policies in order to accommodate various workloads. This was done by adjusting prepage memory allocation, or what they refer to as *target allocation*. The optimum target allocation is determined by a cost-benefit analysis done for all possible target allocations. Prepaged allocation costs and benefits are generated through the use of histograms that track the references to each individual position in the LRU page queue.

This work is essentially the basis for my research. My proposal extends their concept of a dynamic target allocation to a more adaptive design in which all prepage parameters are dynamic.

3. Design and Implementation

Since DAP is an extension to demand prepaging, all designs are targeted for O/S kernels (or VMMs). The implementations of these designs should therefore be as efficient as possible in order to minimize any associated overhead.

3.1.1 Basic Data Structures

In order to implement the concept of dynamic prepage parameters, and even the concept of prepaging itself, several basic data structures are necessary. The first of which is called a *Used Page Queue* (see Figure 3.1). This queue is used to maintain the page numbers of used pages, both resident (pages that are currently loaded in main memory) and non-resident (pages that exist on the backing store only). For prepaged pages, a similar queue, called the *Prepaged Page Queue*, is required (see Figure 3.2). The sum of the number of resident entries in both queues is equal to the number of page frames in main memory, since each entry contains the page number of a page frame in main memory.

When considering the prepaged page queue, it is useful to make a distinction between prepaged allocation, or *target allocation* as we will refer to it from now on, and *consumed allocation*. *Consumed allocation* is the number of main memory page frames within the prepaged allocation that are actually being consumed by prepaged pages. It would be counterproductive if we did not allow used pages to populate unused prepaged page frames, despite the fact that those frames are within the prepaged memory allocation. It is for this reason that we distinguish between target and consumed allocation, so that used pages can populate the remainder of page frames between the consumed and target allocations.

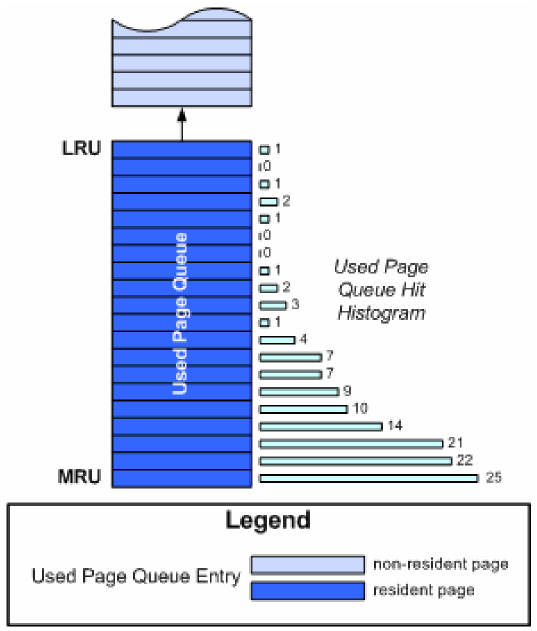


Figure 3.1: Used Page Queue

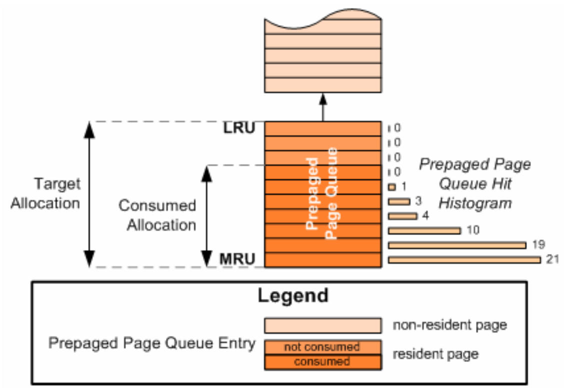


Figure 3.2: Prepaged Page Queue

Both of these queues have hit histograms corresponding to each of their entries. When a page resident in either queue is referenced, the corresponding histogram entry will be incremented. These histograms will be used later to enable the optimization of prepage parameters.

Both queues will also be maintained in a *least recently used (LRU)* ordering. Admittedly, this ordering is impractical with respect to implementation (in hardware) and overhead (in software). For the purposes of this project, however, it will be used to simplify page replacement.

To handle a page reference, the used page queue is scanned to determine if the demanded page is resident. If so, the hit histogram element corresponding to the queue position in which the page number resides will be incremented and the page will be promoted to the most recently used (MRU) position of the queue. If the demanded page is not resident in the used page queue, the prepaged page queue is then checked. If the demanded page is resident in this queue, the hit histogram element corresponding to the queue position in which the page number resides will be incremented. Then the page will be evicted from the prepaged page queue and promoted to the MRU position of the used page queue. If the demanded page is not resident in the prepaged page queue, a page fault has occurred.

As part of handling a page fault, the demanded page along with additional prepaged pages are fetched from disk into main memory. The demanded page number is inserted into the MRU position of the used page queue. If by adding a page reference to the used queue it exceeds its allocation (defined as the difference between the total number of main memory page frames and the prepaged consumed allocation), the LRU page reference of the queue is evicted into the non-resident section of the queue as to reflect the eviction of that page from main memory.

Likewise, all prepaged page numbers fetched from disk are placed in the MRU positions of the prepaged page queue. If by adding these page references to the queue the prepaged target allocation is exceeded, the appropriate number of references are evicted into the non-resident section of the queue, just as the pages they reference are evicted from main memory.

3.1.2 Dynamic Prepaged Allocation

Just as costs and benefits were considered when evaluating the efficacy of prepaging, so too should they be taken into account when trying to optimize prepage allocation. Kaplan et al. [4] proposed a method for calculating the cost and benefit of all possible target allocations as a means for determining which is optimum. This method will be used as part of my proposed DAP system for adapting prepaged allocation.

The cost of a certain target allocation can be defined as the number of page faults that would have occurred had that specific allocation been used. The benefit of a particular target allocation can be defined as the number of page faults that would have been averted if that allocation were implemented. The hit histograms of each queue can be used to calculate these costs and benefits. Assuming a main memory size of *m* page frames, and a target allocation *t*, the cost and benefit of this allocation are defined as follows:

$$Cost(t) = \sum_{i=m-t}^{m} UsedQueueHistogram[i]$$

$$Benefit(t) = \sum_{i=1}^{t} PrepagedQueueHistogram[i]$$

A net reduction in misses can then be calculated as the difference between the benefit and cost of a particular target allocation. The optimum target allocation is finally defined as the allocation that produces the maximum net reduction in misses, as seen below. This entire process is depicted in Figure 3.3.

$$TargetAlloc. = \underset{t}{\operatorname{argmax}}\{Benefit(t) - Cost(t)\}$$

As programs are executed and page references accumulate, it is clear that the hit histograms will saturate in the sense that their values will no longer reflect the "current" behavior or phase of a program. Since these histogram values are an integral part of optimizing prepage allocation, they must be conditioned such that they are more reflective of a program's recent behavior. Kaplan et al. [4] avoided histogram saturation by periodically decaying all entries by a constant value λ, $0 < \lambda < 1$, such that a histogram entry after decay would equal its value before decay times the decay value. Surprisingly, their analysis found that the value of the decay variable had little to no effect on performance.

Both the cost-benefit analysis and the histogram decay are computationally intensive. The only way to minimize the overhead associated with these actions is to hide it by performing the computations during the disk accesses associated with the handling of a page fault. Since disk latency is many orders of magnitude higher than processor clock cycles, these calculations should not present any additional latency if carried out while pages are being fetched from the backing store. However, this does not imply that these processes do not require efficient implementation. Other tasks can be scheduled during disk accesses, such as execution of different threads in the processor, while waiting on the demanded page to be loaded. Another way to minimize the overhead of the dynamic prepage allocation calculations is to force the parameter to be updated less often. Instead of updating prepage allocation upon every page fault, updates can be performed on a much longer period, thereby reducing the computational overhead.

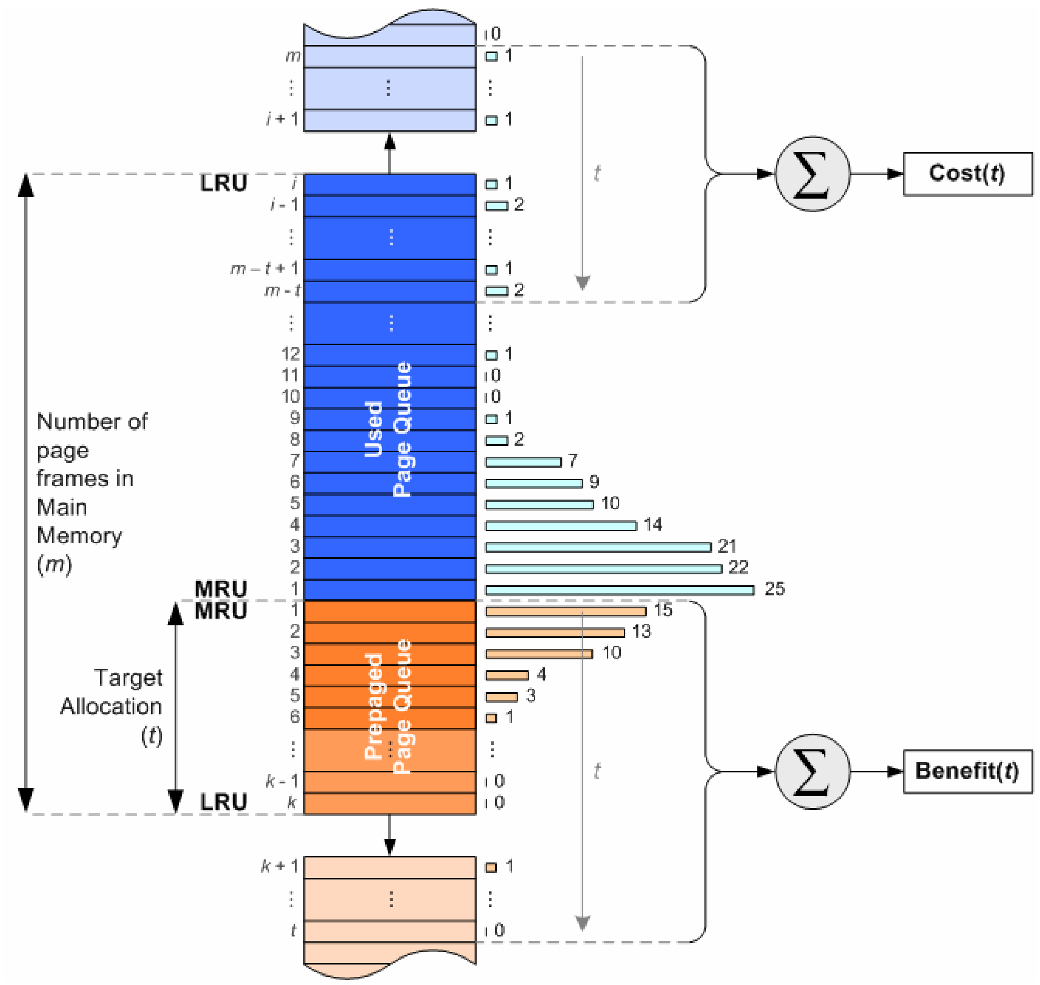


Figure 3.3: Prepage Allocation Cost-Benefit Analysis

3.1.3 Dynamic Degree

The degree of prepaging is a parameter that is closely related to the method used to predict prepages. If prepaged pages are being referenced soon after being fetched, than the prediction method is performing effectively. If this is the case, it would be most beneficial if prepage degree is high, so that more pages can be prepaged, and more page faults can be avoided. Conversely, if prepage prediction is not effective, the optimum prepage degree would be relatively low so as to minimize wasteful memory consumption.

In DAP, this relationship is exploited in order to optimize prepaging degree. To do so, a new data structure called the *Degree Queue* is required (see Figure 3.4). This is a simple three-entry queue with a corresponding hit histogram. For a degree *d*, the *d*th, (*d*+1)th, and (*d*+2)th prepage predictions are stored in the degree queue after every page fault. It is important to note that although prepage page prediction numbers *d*+1 and *d*+2 are stored in this queue, the page which those predictions reference are *not* actually loaded into physical memory.

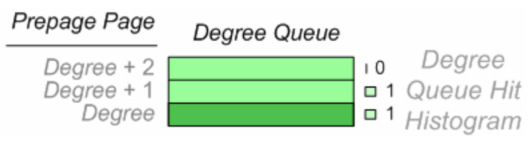


Figure 3.4: Degree Queue

The degree queue hit histogram enables the system to determine whether prepaging additional pages under the current prediction method would be beneficial. If the histogram shows there were references to prepage page prediction number *d*+2, then the degree will be promoted to *d*+2. Likewise, if prediction number *d*+1 was referenced, the degree will be incremented by one. Otherwise, if the *d*th prepage page prediction was referenced, than the degree is left unchanged. If none of these predictions were referenced since the last page fault, then the degree can actually be decremented, since the current prediction method is not performing accurately. Figure 3.5 depicts this algorithm for dynamically adapting prepage degree.

Pre-defined, static minimum and maximum prepage degrees should also be implemented in order to avoid what could be considered *runaway adaptation*, a scenario in which the degree is constantly incremented or decremented to the eventual detriment of the system.

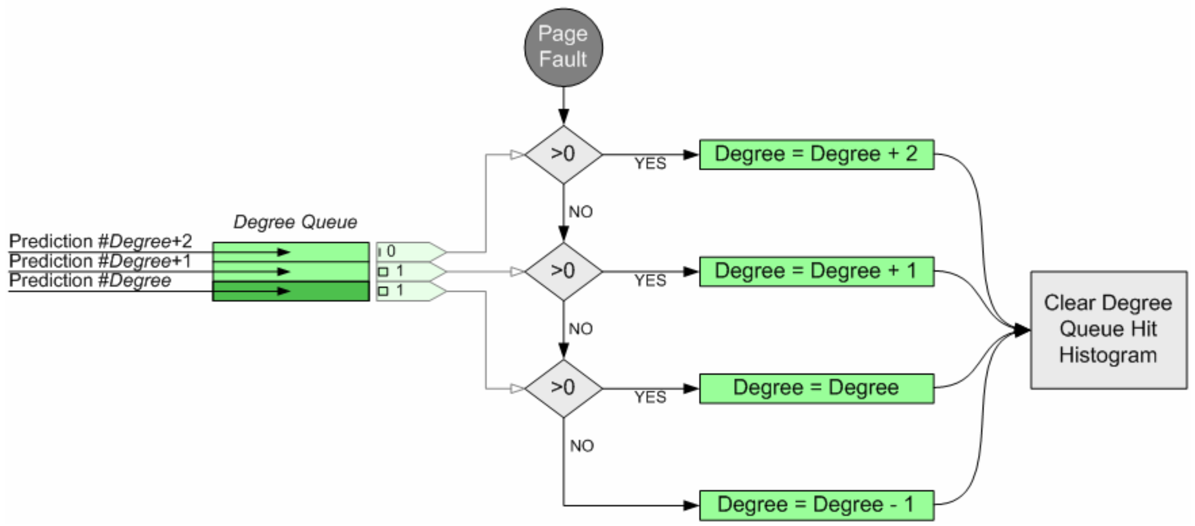


Figure 3.5: Dynamic Optimization of Prepage Degree

3.1.4 Dynamic Prepage Prediction

There are several prepage prediction schemes, each performing better in different scenarios. Three prediction methods are used in the proposed DAP system:

1. **Address-local Prediction**: Predicts pages nearby in the address space relative to the page being demanded are likely to be referenced soon.

   Example: Referenced page number n
   Predictions: $n + 1, n - 1, n + 2, n - 2, \ldots$

2. **Recency-local Prediction**: Predicts pages nearby in an LRU ordering to the page being demanded are likely to be referenced soon.

   Example: Reference to page found at LRU queue position p
   Predictions: Pages found in LRU queue positions
   $p - 1, p + 1, p - 2, p + 2, \ldots$

3. **Stride Prediction**: Predicts pages located at a constant stride in the address space from the page in demand relative to the MRU page are likely to be referenced soon.

   Example: Degree d, Referenced page number n, MRU page queue entry $p \rightarrow$ stride $s = n - p$
   Predictions: $n + s, n + 2s, n + 3s, \ldots, n + ds$

In order to dynamically choose the optimum prediction method, a new data structure must be introduced. Figure 3.6 shows a *Prediction Queue*, a queue which holds the prepage page predictions associated with a prediction method. This queue also has a corresponding hit histogram which will be used to determine the optimum prediction method.

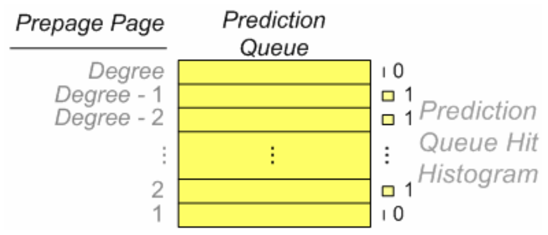


Figure 3.6: Prediction Queue

The optimum prediction method can be easily defined as the method that predicted pages that resulted in the most references. This optimization can be implemented using the prediction queue hit histograms. Whichever prediction method has the greatest sum of hit histogram entries since the last page fault is the optimum prediction method. Figure 3.7 depicts this optimization process.

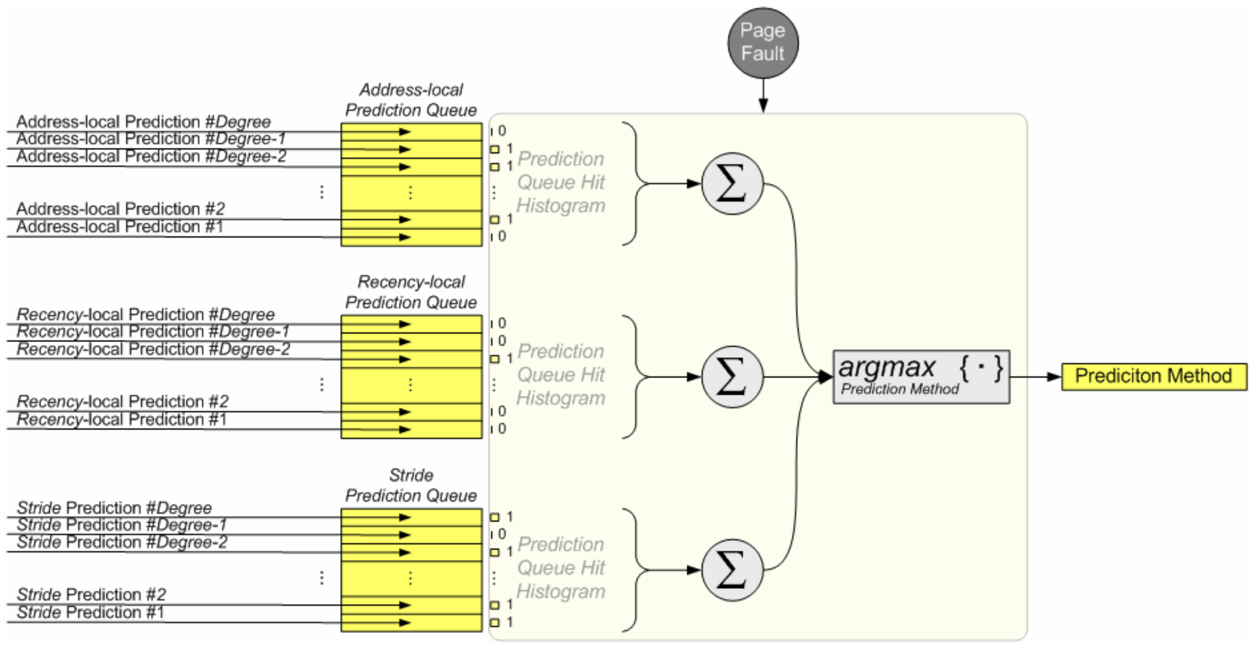

Page Fault
Address-local Prediction Queue
Address-local Prediction #Degree
Address-local Prediction #Degree-1
Address-local Prediction #Degree-2
Address-local Prediction #2
Address-local Prediction #1
Recency-local Prediction Queue
Recency-local Prediction #Degree
Recency-local Prediction #Degree-1
Recency-local Prediction #Degree-2
Recency-local Prediction #2
Recency-local Prediction #1
Stride Prediction Queue
Stride Prediction #Degree
Stride Prediction #Degree-1
Stride Prediction #Degree-2
Stride Prediction #2
Stride Prediction #1
Prediction Queue Hit Histogram
Σ
argmax {·}
Prediction Method
Prediciton Method

Figure 3.7: Dynamic Optimization of Prepage Prediction Method

4. Simulation Methodology

To test and evaluate the efficacy of dynamically adaptive prepaging, its policies must be applied to a memory reference trace. The performance results (page fault rate, disk transfers, etc.) may then be compared to the performance of a completely static demand prepaging policy. To enable such a performance comparison, a trace-driven simulator was developed in C++.

4.1 DAPsim

DAPsim is a trace-driven demand prepaging simulator developed to enable the design, testing, and evaluation of DAP concepts. *DAPsim* has the following capabilities:

1. Maintains main memory page queues
2. Handles page references
3. Models dynamic prepage allocation, degree, and prediction method
4. Accumulates performance statistic (page faults, fault rate, reference trace characteristics, hit/miss trace, disk transfers, etc.)

The simulator reads through a memory reference trace, handling each page reference one at a time. The state of the prepaging system after handling every reference is exactly that of a real-world implementation.

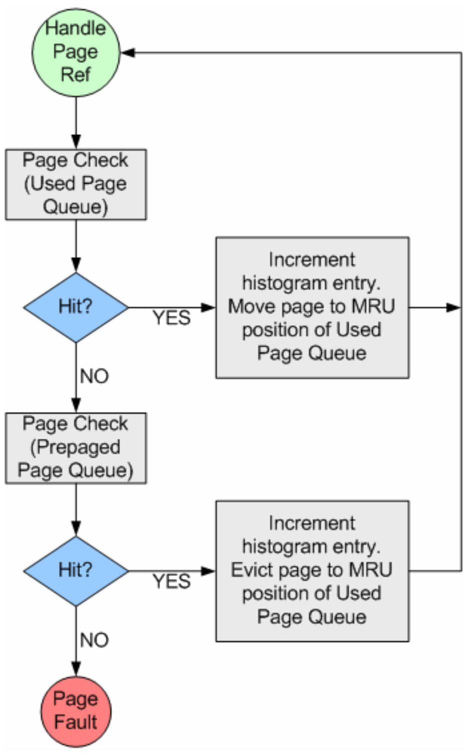

Handle Page Ref
Page Check (Used Page Queue)
Hit?
YES
Increment histogram entry. Move page to MRU position of Used Page Queue
NO
Page Check (Prepaged Page Queue)
Hit?
YES
Increment histogram entry. Evict page to MRU position of Used Page Queue
NO
Page Fault

Figure 4.1: Page Reference Flow

Figures 4.1 and 4.2 show the flow of the *DAPsim* simulator. To handle a page reference, the simulator first checks if the reference is resident in the used page queue. If so, the corresponding hit histogram entry is incremented and the reference is promoted to the MRU position of the queue. If the page reference is not resident in the used queue, the prepage page queue is checked. If the reference is resident in the prepaged page queue, the corresponding hit histogram entry is incremented and the reference is evicted from the prepaged queue into the MRU position of the used queue. If it is not resident, then a page fault has occurred.

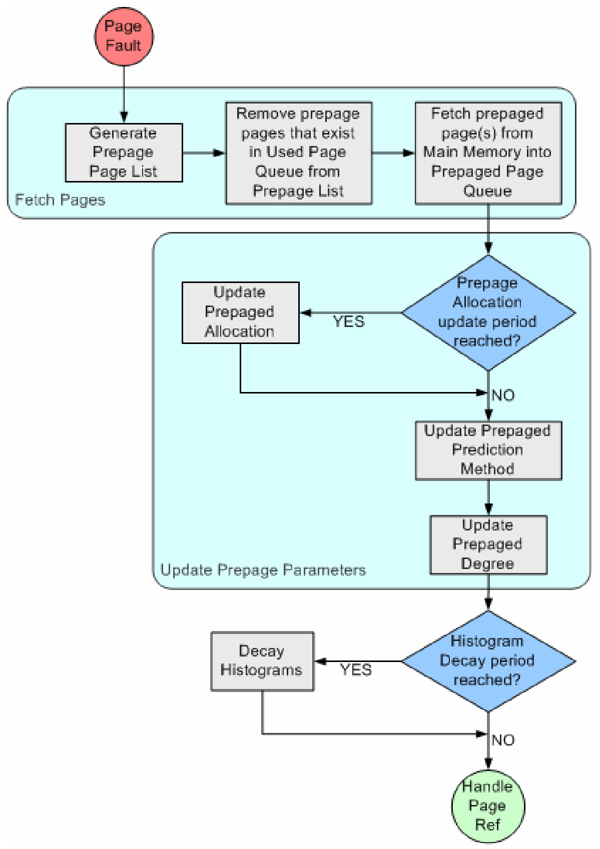

Page Fault
Generate Prepage Page List
Remove prepage pages that exist in Used Page Queue from Prepage List
Fetch prepaged page(s) from Main Memory into Prepaged Page Queue
Fetch Pages
Update Prepaged Allocation
YES
Prepage Allocation update period reached?
NO
Update Prepaged Prediction Method
Update Prepaged Degree
Update Prepage Parameters
Decay Histograms
YES
Histogram Decay period reached?
NO
Handle Page Ref

Figure 4.2: Page Fault Flow

To handle a page fault, *DAPsim* “fetches” the demanded page along with several prepaged pages from main memory. The reference to the demanded page is placed in the MRU position of the used page queue. The references to the prepaged pages are place in the MRU positions of the prepaged page queue. Then, the prepage parameters are updated according to the previously described methods. Obviously, in the real-world implementation of this system, prepage parameters would be updated in parallel with the accesses to the backing store so as to hide the computational overhead. Since *DAPsim* does not directly simulate any cycle-accurate latencies, updating prepage parameters immediately after fetching pages from disk is an acceptable model of the system.

After prepage parameters are updated, several other maintenance-related tasks are performed, such as decaying hit histograms if necessary. Once this is done, the next page reference in the memory trace is handled, thus completing the cycle.

4.2 Simulation Methodology

In order to properly evaluate the performance of my proposed DAP system, a broad range of memory reference behavior must be tested. Simulations were performed on several uni-programmed memory reference traces over a range of main memory sizes. The reference traces were gathered using the *Etch* instrumentation tool on a Windows NT system. The majority of the traces used are the same used by Kaplan et al. [4] in their research on adaptive caching for demand prepaging. The traces include a mix of batch-style processes (gcc, compress), synthetic processes (sawtooth, lu, mm16, and mm32), and interactive, GUI processes (Acrobat Reader, Go, Netscape Navigator, Photoshop, PowerPoint, and Word). Table 4.1 shows reference trace details.

Table 4.1: Memory Reference Trace Descriptions

Trace Name	Benchmark Type	Program	Program Description
acroread	Real	Acrobat Reader / WinNT	PDF viewer
cc1	Real	Compiler	Batch-style process
compress95	Real	Compress	Batch-style compression
sawtooth	Synthetic	Custom Sawtooth	Sawtooth trace
go	Real	GO	Interactive GUI process
lu	Micro	LU	LU matrix decomposition
mm16	Micro	Matrix Multiply (16x16)	Matrix Multiply (16x16)
mm32	Micro	Matrix Multiply (32x32)	Matrix Multiply (32x32)
netscape	Real	Netscape / WinNT	Web browser
powerpoint	Real	Power Point / WinNT	Slide show developer
winword	Real	Word / WinNT	Word processing

The use of such a variety of benchmarks was done in an attempt to gauge performance over different reference patterns. Figures 4.3 through 4.5 show the variation in memory reference profiles used to evaluate DAP performance. The memory reference profiles for all the traces used in the evaluation process are located in Appendix A.

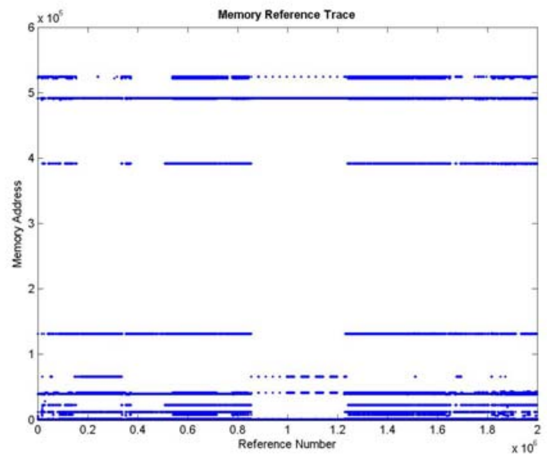


Figure 4.3: Netscape Memory Reference Profile

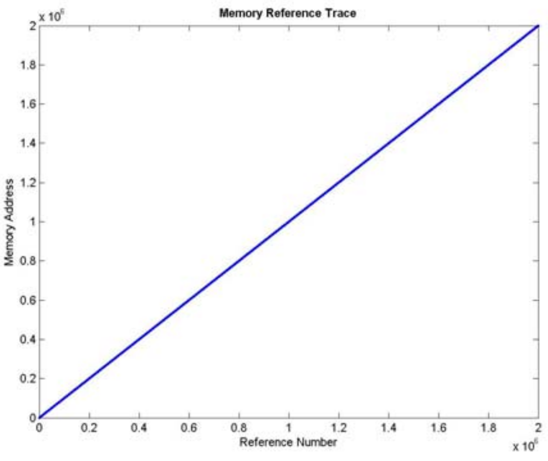


Figure 4.4: Sawtooth Memory Reference Profile

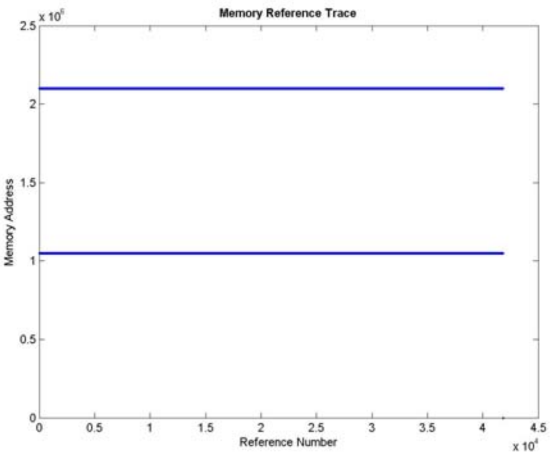


Figure 4.5: LU Decomposition Memory Reference Profile

Different system configurations were simulated in order to evaluate the performance relationships between different prepage parameters as well as DAP as a whole. Table 4.2 shows the different modes that were simulated for each memory reference trace. Obviously, *DAP Mode* is the mode which will be used to evaluate the performance of my proposed DAP concepts.

Table 4.2: Simulation Modes

	Prepage Allocation	Degree	Prediction Method
Static Prepage Parameter Mode	Static	Static	Static
Dynamic Prepage Allocation (DPA) Mode	*Dynamic*	Static	Static
Dynamic Degree (DD) Mode	Static	*Dynamic*	Static
Dynamic Prediction Method (DPM) Mode	Static	Static	*Dynamic*
DAP Mode	*Dynamic*	*Dynamic*	*Dynamic*

4.3 *System Settings*

It is important to note the system settings used during simulations. Table 4.3 shows the values of all major system parameters. The values of these parameters were defined after careful analysis of empirical data obtained through simulation. Values were chosen such that performance was optimized in static parameter mode. This was done so that DAP could be evaluated against a formidable control.

Table 4.3: System Settings

	Parameter	Value	Unit
Used Page Queue	Initial Used Queue Size	$0.90 \times \left(\text{Main Mem. Page Frames}\right)$	Queue Entries
	Minimum Used Queue Size	$0.25 \times \left(\text{Main Mem. Page Frames}\right)$	Queue Entries
Prepage Allocation	Initial Prepage Target Allocation	$0.20 \times \left(\text{Main Mem. Page Frames}\right)$	Queue Entries
	Maximum Prepage Target Allocation	$0.75 \times \left(\text{Main Mem. Page Frames}\right)$	Queue Entries
	Minimum Prepage Target Allocation	$0.20 \times \left(\text{Main Mem. Page Frames}\right)$	Queue Entries
	Prepage Allocation Update Period	5	Page Fault Events
	Prepage Allocation Initialization Period	200,000	Page References
Prepage Degree	Initial Degree	5	Pages
	Maximum Degree	$0.75 \times \left(\text{Main Mem. Page Frames}\right)$	Pages
	Minimum Degree	5	Pages
Histogram Decay	Decay Period	2,000,000	Page References
	Decay Value	1/2	-
Trace Simulation	Sample Size	2,000,000	Page References

5. Simulation Results

Many useful statistics were gathered during simulations, the most important of which, for the purpose of a performance evaluation, are the quantity of page faults and disk transfers. When comparing to a demand prepaging policy with completely static parameters, it is more useful to relate corresponding statistics to determine whether or not the proposed concepts offer any performance gains. Therefore, the simulation results will be focused on two major statistics, *page fault reduction* and *page transfer increase*.

5.1 *Page Fault Reduction*

The ultimate goal of dynamically adaptive prepaging is to reduce the page fault rate over a broad range of workloads. Simulations were conducted on the previously mentioned memory reference traces over a range of main memory sizes in order to determine page fault rates. Then, page faults rates for the various simulation modes (see Table 4.2) were compared to a control (*Static Prepage Parameter* mode) in a statistic called *page fault reduction*. Page fault reduction is computed as follows:

$$PageFaultReduction_{Mode\_X} \quad \frac{PageFaultRate_{StaticMode} \quad PageFaultRate_{Mode\_X}}{PageFaultRate_{StaticMode}},$$

where $PageFaultReduction_{Mode\_X}$ is the reduction in page fault rate by Mode X in relation to Static Prepage Parameter mode. Table 5.1 and Figure 5.1 show DAP page fault reductions corresponding to each memory reference trace.

Table 5.1: DAP Page Fault Reduction

	DAP Page Fault Reduction (per Main Mem. Size in Pages)					
Benchmark	**500**	**1000**	**2000**	**5000**	**10000**	***Average***
acroread	-6.80%	5.46%	1.36%	0.91%		**0.23%**
cc1	5.32%	5.32%	11.70%	11.70%		**8.51%**
compress	9.26%	10.19%	12.96%	13.89%		**11.57%**
sawtooth	95.86%	97.64%	98.23%	97.52%		**97.31%**
go	-36.78%	-5.88%	-5.88%	-5.88%		**-13.61%**
lu	74.55%	69.57%	69.57%	69.57%	69.57%	**70.56%**
mm16	62.69%	64.93%	64.93%	64.93%		**64.37%**
mm32	40.77%	16.69%	79.38%	79.62%	79.62%	**59.22%**
netscape	-7.12%	7.10%	2.37%	-1.08%		**0.32%**
powerpoint	2.96%	0.80%	0.00%	0.00%		**0.94%**
winword	-5.45%	8.00%	4.53%	0.40%	2.39%	**1.97%**
Average	**21.39%**	**25.44%**	**30.83%**	**30.14%**	**50.53%**	**31.66%**

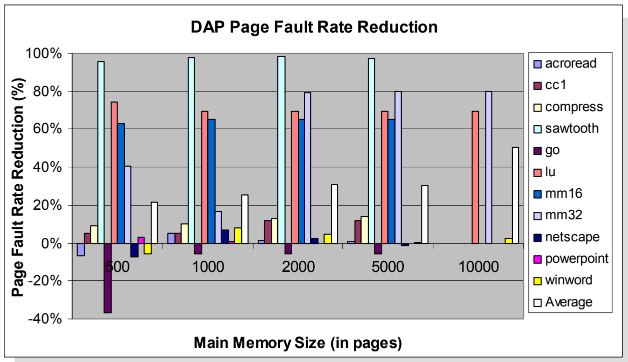


Figure 5.1: DAP Page Fault Reduction

The best reduction was is the *sawtooth* benchmark, most likely because of its extremely predictable memory references. On average, simulations showed a 32% reduction in page fault rate across all benchmarks.

The majority of this reduction was in the synthetic and micro benchmarks, again most likely because of the predictability of their memory references. In real benchmarks, the average page fault reduction was between 1.5 and 4%, depending on the benchmarks included in the average.

The other simulation modes showed a wide variation of page fault reduction. Figures 5.2 through 5.4 show page fault reductions for each of these modes. The highest average page fault reduction, approximately 36%, was under dynamic degree mode.

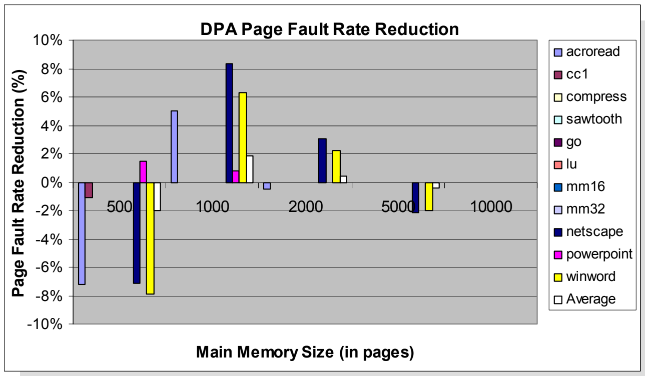


Figure 5.2: DPA Page Fault Reduction

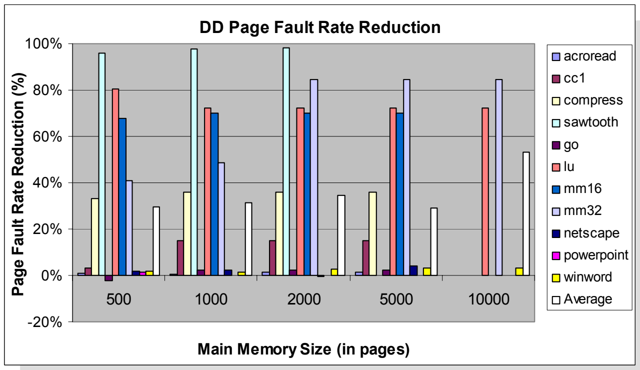


Figure 5.3: DD Page Fault Reduction

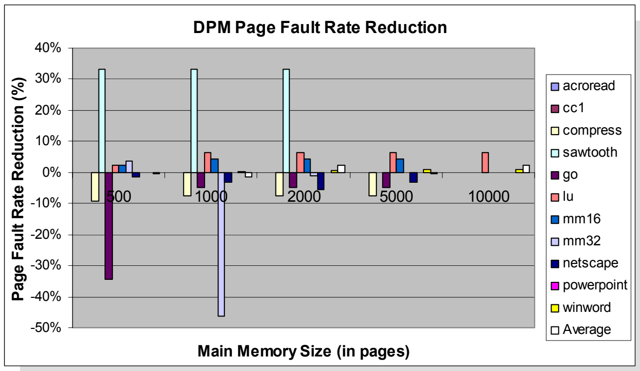


Figure 5.4: DPM Page Fault Reduction

5.2 *Page Transfer Increase*

Another important statistic that must be considered when evaluating the efficacy of a prepaging scheme is page transfers. It is obvious that demand prepaging will lead to an increase in page transfers when compared to demand paging. The question this research aims to answer is whether *dynamic* demand prepaging (namely my proposed dynamically adaptive prepaging scheme) will likewise instigate an increase in page transfers.

5.2.1 Total Page Transfers

It is useful to define a comparative statistic in order to more easily evaluate the impact DAP has on page transfers relative to a static demand prepaging scheme. This new statistic, called *page transfer increase*, is defined as follows:

$$PageTransferIncrease_{Mode\_X} \quad \frac{PageTransfers_{Mode\_X} \quad PageTransfers_{StaticMode}}{PageTransfers_{StaticMode}},$$

where $PageTransferIncrease_{Mode\_X}$ is the percent increase in page transfers of Mode *X* relative to static demand prepaging.

Table 5.2 and Figure 5.5 show DAP page transfer increases in all benchmarks over a range of memory sizes. The average page transfer increase across all benchmarks and memory sizes was approximately 20%. Again, the highest page transfer increases happened to occur in the micro and synthetic benchmarks. Among the real benchmarks, the average page transfer increase was approximately 12%.

Table 5.2: DAP Page Transfer Increase

	DAP Page Transfer Increase (per Main Memory Size in Pages)					
Benchmark	**500**	**1000**	**2000**	**5000**	**10000**	***Average***
acroread	8.79%	-0.80%	2.35%	2.87%	2.37%	**3.12%**
cc1	15.10%	17.94%	12.69%	12.69%		**14.61%**
compress	34.54%	33.73%	30.32%	29.52%		**32.03%**
sawtooth						
go	45.33%	15.38%	15.38%	15.38%		**22.87%**
lu	14.26%	23.27%	23.27%	23.27%	23.27%	**21.47%**
mm16	29.06%	27.08%	27.08%	27.08%		**27.57%**
mm32	34.88%	156.26%	35.93%	33.59%	33.59%	**58.85%**
netscape	12.91%	1.39%	5.18%	8.59%		**7.02%**
powerpoint	-0.14%	0.59%	1.40%	1.25%		**0.78%**
winword	9.02%	-2.52%	1.15%	4.51%		**3.04%**
Average	**20.38%**	**27.23%**	**15.48%**	**15.88%**	**19.74%**	**19.74%**

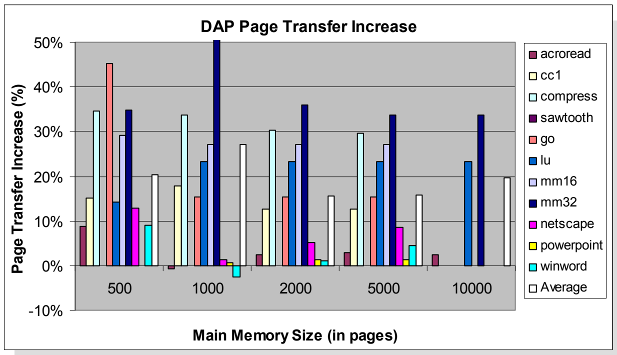


Figure 5.5: DAP Page Transfer Increase

5.2.2 Average Page Transfers

Perhaps a more significant statistic is the increase in page transfers per page fault. This metric is certainly expected to be higher relative to a simple demand paging scheme, but how does a dynamic demand prepaging system compare to a static one? Table 5.3 and Figure 5.6 show average increases in page transfers per page fault under DAP for a variety of benchmarks, over a range of memory sizes. Page transfers per page fault increased by an average of 4.32 pages per fault across all benchmarks and memory sizes. Again, the bulk of this increase was seen in the micro and synthetic benchmarks. Among the real benchmarks, the average increase in page transfers per page fault was approximately 0.70. Average page transfers per page fault in static demand prepaging was calculated to be 4.77, whereas the average for DAP was 9.72.

Table 5.3: DAP Increase in Page Transfers per Page Fault

	DAP Average Page Transfer Increase (per Main Memory Size in Pages)					
Benchmark	**500**	**1000**	**2000**	**5000**	**10000**	***Average***
acroread	0.10	0.26	0.20	0.21		**0.19**
cc1	1.05	1.19	1.34	1.34		**1.23**
compress	2.23	2.25	2.29	2.32		**2.27**
sawtooth						
go	0.32	0.45	0.45	0.45		**0.42**
lu	14.21	12.44	12.44	12.44		**12.89**
mm16	10.17	10.84	10.84	10.84		**10.67**
mm32	5.67	8.44	22.61	22.46		**14.80**
netscape	0.27	0.47	0.41	0.39		**0.38**
powerpoint	0.16	0.08	0.08	0.07		**0.10**
winword	0.17	0.31	0.31	0.26		**0.26**
Average	**3.43**	**3.67**	**5.10**	**5.08**		**4.32**

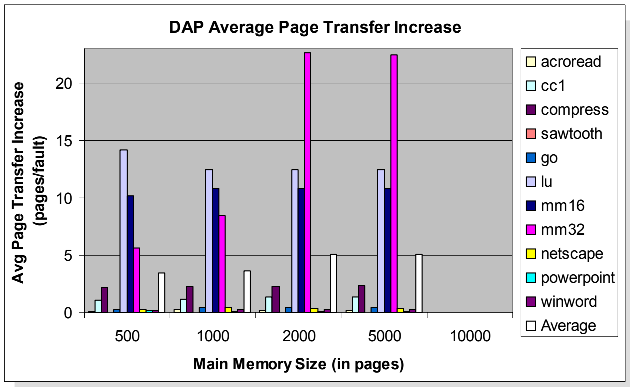


Figure 5.6: DAP Increase in Page Transfers per Page Fault

6. Evaluation

The simulation data presented in the previous section show two main results: 1) that the proposed DAP policies reduced average page fault rates across a variety of memory reference traces, and 2) DAP also leads to an increase in page transfers, specifically average page transfer rates. In order to evaluate the performance of DAP, both of these results must be weighed against each other.

6.1.1 Page Fault Reduction vs. Page Transfer Increase

In a direct comparison, the 32% reduction in page faults outweighs the 20% increase in page transfers by a significant amount. This comparison can be misleading if not properly examined. Although an increase in page transfers, in general, degrades performance, the increases seen in DAP can mostly be attributed to an increase in average page transfer rate (average page transfers per page fault). So, although page transfers have *increased*, page faults have *decreased*. In fact, page transfers per page fault increased by an average of 4.32 pages, or approximately 104% relative to static demand prepaging. This means that DAP requires approximately 100% more disk bandwidth than static demand prepaging.

Although this may seem alarming, one must consider and compare the implications of decreased disk latency at the expense of increased bandwidth utilization in order to fully evaluate the reported results. As previously mentioned, disk bandwidths are increase at about four times the rate in which disk latencies are decreasing on a year-by-year basis. This means that reductions in disk access events should be weighed more heavily than increased utilization of disk bandwidth in a comparative evaluation of performance enhancement. It is important to note, however, that the effect of an increase in average page transfer rate on disk latency is highly dependant on the degree in which pages can be clustered on the disk. Page clustering on storage disks is a widely researched topic that will not be explored in this paper.

6.1.2 The Effects of Main Memory Size

As Figure 5.1 shows, the average page fault reduction across all benchmarks increased, to an extent, with main memory size. It is intuitive that page faults are likely to decrease under any paging policy as main memory size increases. It is not so intuitive, however, that page fault *reduction* – a measure of the degree to which page faults are reduced relative to a static prepaging policy – increases with main memory size, as was shown. It is undoubtedly a desirable result nonetheless.

Just as desirable of an attribute of DAP is the fact that no such trend exists with average page transfer increase. As Figure 5.5 shows, average page transfer increase did not grow with main memory size.

6.1.3 Overall Performance

The overall performance of my proposed DAP scheme met my expectations. It was evident that the prepage parameters were being dynamically adapted to exploit trace predictability and resource state. Figures 6.1 and 6.2 show an example of DAP's demonstrated ability to adapt to trends in the memory reference stream. In Figure 6.1, a constant density of page faults can be observed throughout the execution of the *sawtooth* reference trace. In Figure 6.2, however, DAP is successfully able to adapt its prepage parameters to reduce page faults. In fact, page fault reduction only increases as the trace continues to execute.

As previously mentioned, the average page fault reduction for real benchmarks was observed to be approximately 1.5%. This includes the *go* memory trace in which DAP actually caused an increase in page faults. The *go* benchmark had an average page fault rate of 4.275×10^{-5} faults per reference under static demand prepaging, but an average of 4.863×10^{-5} under DAP, an increase of 13.74%. The *go* benchmark was the *only* benchmark used in this project in which the application of DAP actually increased the average page fault rate. This demonstrates DAP's potential for an undesired impact on virtual memory management.

Despite this anomalous increase in page faults, a 1.5% average reduction in page fault rate should not be considered insignificant. When page references can reach practically endless numbers, during the indefinite execution of one or many workloads, an average page fault rate reduction of 1.5% translates to a substantial enhancement.

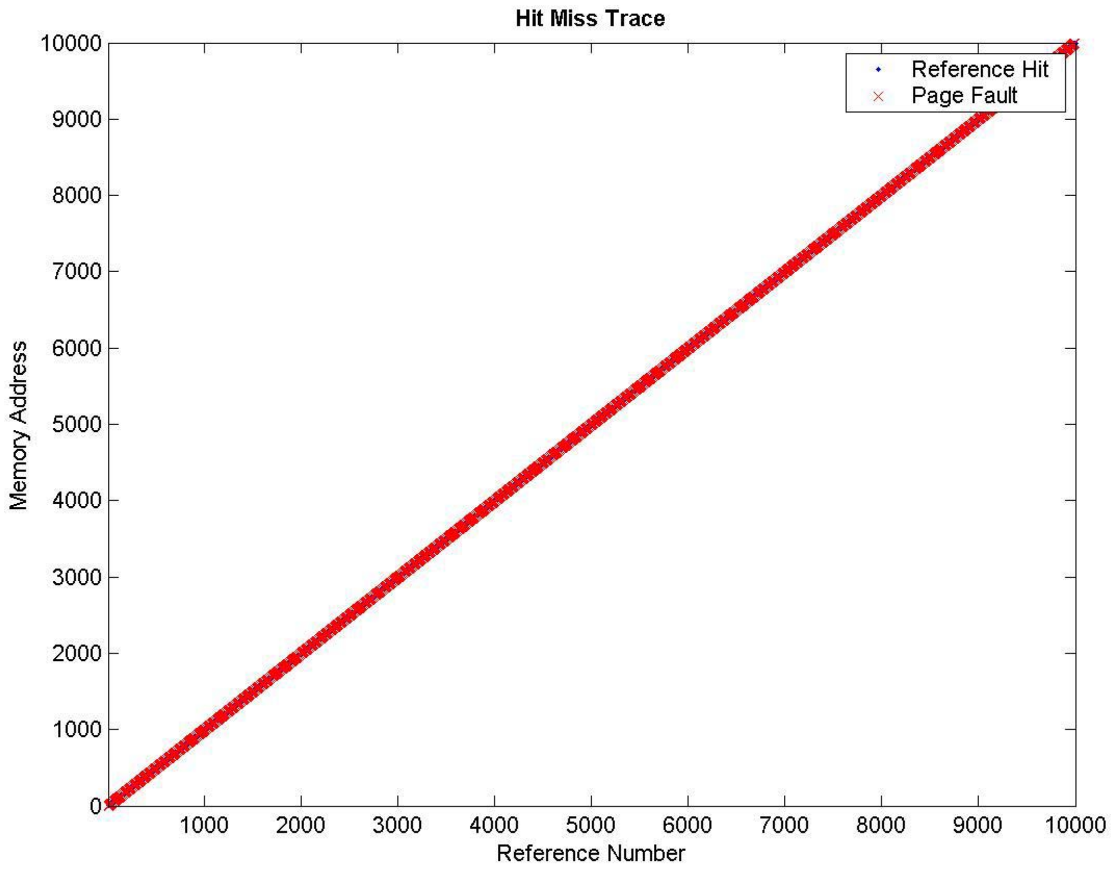


Figure 6.1: Memory Reference Trace Results under Static Parameter Mode

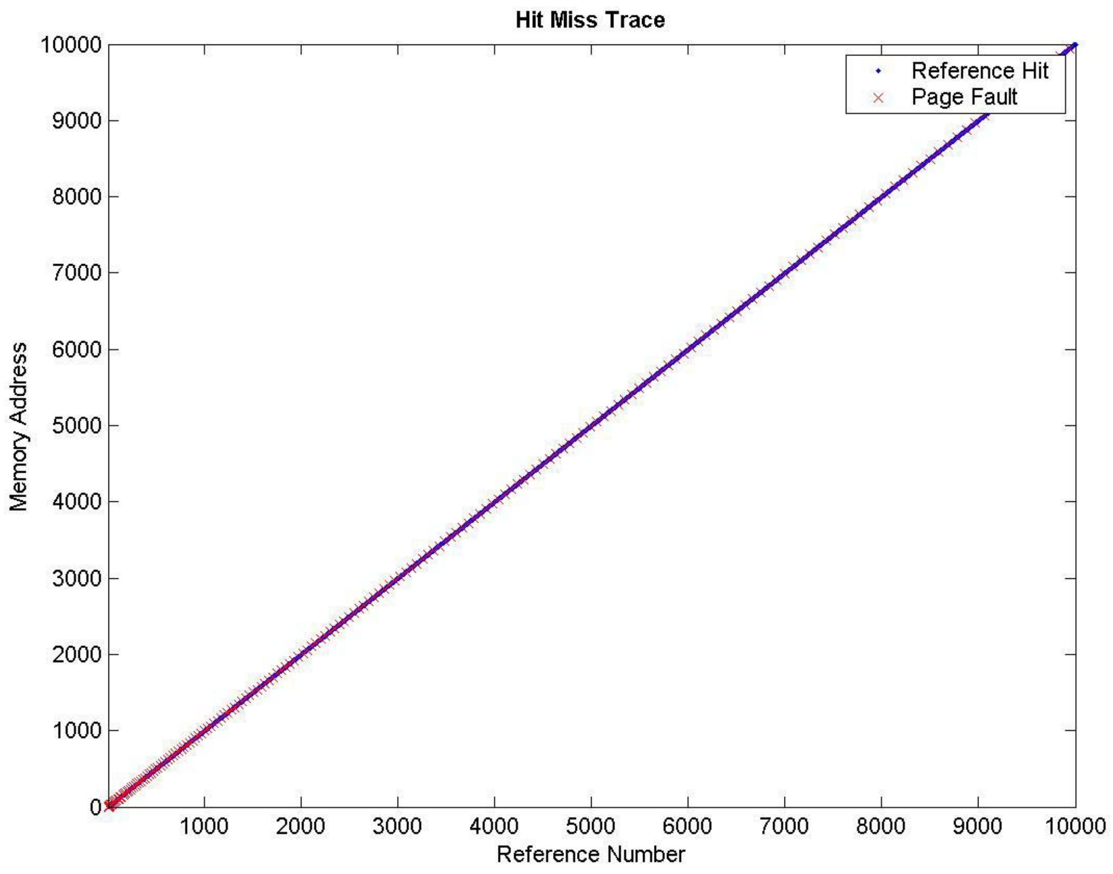


Figure 6.2: Memory Reference Trace Results under DAP

7. Conclusion

The target environment for dynamically adaptive prepaging is not one with a single executing program, however. So, in a sense, the simulations performed on uni-programmed memory reference traces fail to demonstrate (at least directly) any potential benefits DAP concepts would present in a more real-world scenario of multi-programmed or VM workloads. In fact, the simulation results seem to show a correlation between memory reference trace *range* and average page fault rate reduction that might suggest that DAP would not perform as well in multi-programmed environments.

Table 5.1 shows that DAP results in an average page fault rate reduction of 1.97% for the *winword* benchmark, yet only a 0.32% average reduction for the *netscape* trace, a trace with a larger range of memory references. Since it is likely that multi-programmed and VM workloads would have a much higher reference range relative to a uni-programmed workload, this data suggests that, if this trend is maintained, DAP-induced page fault rate reduction would not be nearly as significant in the multi-programmed and VM environments.

Despite what empirical data might suggest, the significant performance enhancements seen in the simulation results warrant a further investigation into DAP's performance with multi-programmed and VM workloads. Although DAP increases average page transfer rates, this can be effectively neglected with increasing disk bandwidths. What is not negligible is the reduction in page fault-induced disk accesses enabled by dynamically adaptive prepaging. Avoiding even the smallest amount of disk accesses provides significant performance enhancement.

Finally, it is in my judgment that the overhead and complexity associated with the necessary kernel-level DAP implementations do not negate the benefits of the reduction in disk accesses. Still, more efficient implementations would of course be favorable.

8. Future Work

The results of this research have not only shown the costs and benefits of dynamically adaptive prepaging, they have also provided insight into what may be necessary for more effective concepts. The following are areas for possible expansion to the concept of DAP in an effort for a more effective prepaging system.

8.1 Degeneration–resistant Adaptation

Perhaps the most detrimental consequence of dynamically adaptive prepaging is its potential to degenerate performance relative to a static demand prepaging (or demand paging) scheme. If safeguards could somehow be integrated into a DAP scheme that would limit, if not eliminate, the potential for performance degradation, then such a scheme would be vastly superior to any current proposals.

Such a scheme is, to a certain extent, counterproductive. For it is the vary attributes of the system that are made dynamic to provide the potential for performance improvement that also provide the potential for degradation. Still, it seems viable to develop a dynamic prepaging scheme that at least limits the potential for performance degeneration while still maintaining the benefits of dynamic parameter adaptation.

8.2 Compulsory Fault Prevention

The simulation results presented in this paper show that a significant portion of the page faults under a dynamically adaptive prepaging scheme (or any paging scheme for that matter) are compulsory – page faults that occurred because the demanded pages had never been referenced prior to the fault. Figure 8.1 shows the DAP page fault trace for the *winword* benchmark. Notice the preponderance of page faults near the start of the trace simulation.

Cache hierarchies mitigate such misses through *prefetching*, or the fetching of cache blocks prior to being referenced in anticipation that they soon will be. This concept is somewhat similar to that of prepaging. Prepaging differs from prefetching in a slight but significant detail however. Prepaging is conceptually a reactionary tactic, whereas prefetching is more proactive. Although the incredibly high latency of disk accesses makes a compelling argument against any sort of proactive prepaging policy, the potential for a significant reduction in page faults is reason enough for at least a cursory feasibility analysis.

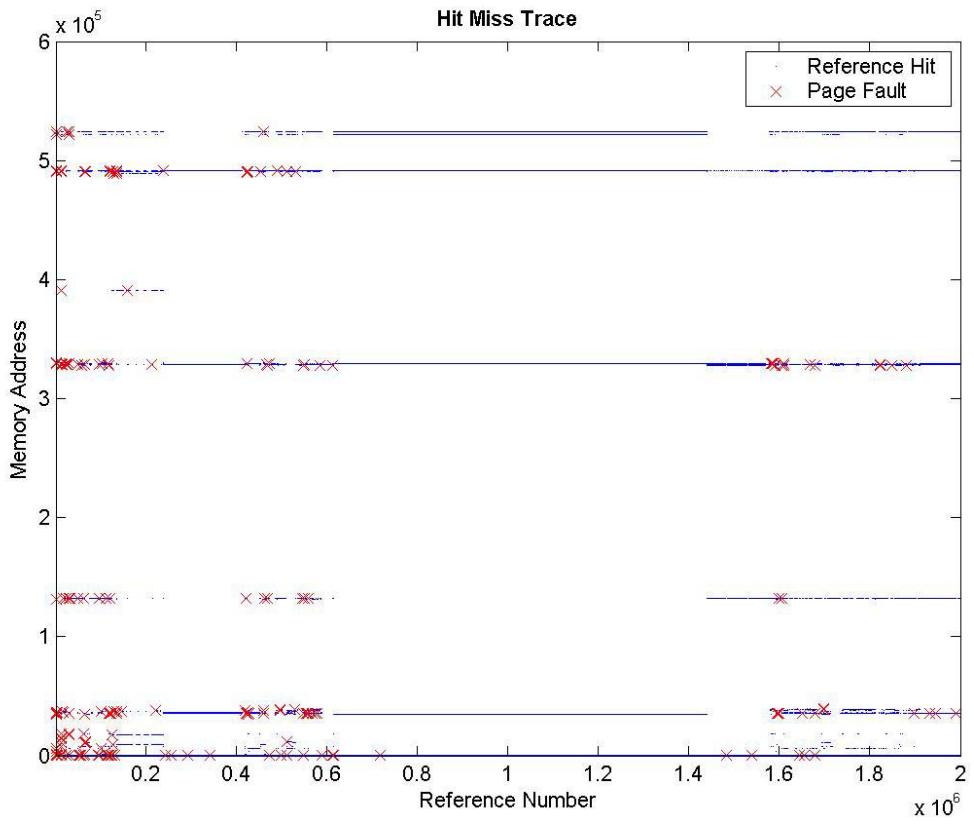


Figure 8.1: ***winword*** **Page Reference Trace**

9. References

[1] M. Joseph. An analysis of paging and program behavior. Computer Journal, 13:48-54, 1970.

[2] A. J. Smith. Sequential program prefetching in memory hierarchies. IEEE Computer, 11(12):7-21, Dec. 1978.

[3] R. N. Horspool and R. M. Huberman. Analysis and development of demand prepaging policies. Journal of Systems and Software, 7:183-194, 1987.

[4] S. F. Kaplan, L. A. McGeoch, and M. F. Cole. Adaptive caching for demand prepaging. In Proceedings of the third international symposium on Memory management, 2002.

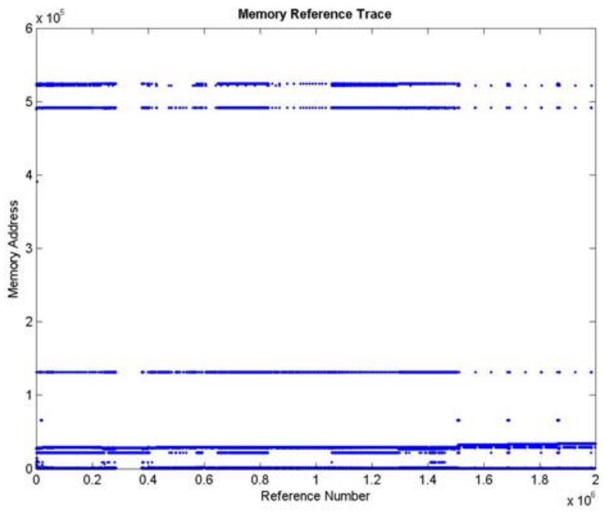
[5] P. Zhou, V. Pandey, J. Sundaresan, A. Raghuraman, Y. Zhou, and S. Kumar. Dynamic Tracking of Page Miss Ratio Curve for Memory Management. ACM SIGARCH Computer Architecture News, 2004

[6] D. C. Lee, P. J. Crowley, J. L. Baer, T. E. Anderson, and B. N. Bershad. Execution characteristics of desktop applications on Windows NT. In 25th Annual International Symposium on Computer Architecture. IEEE Computer Society Press, 1998.

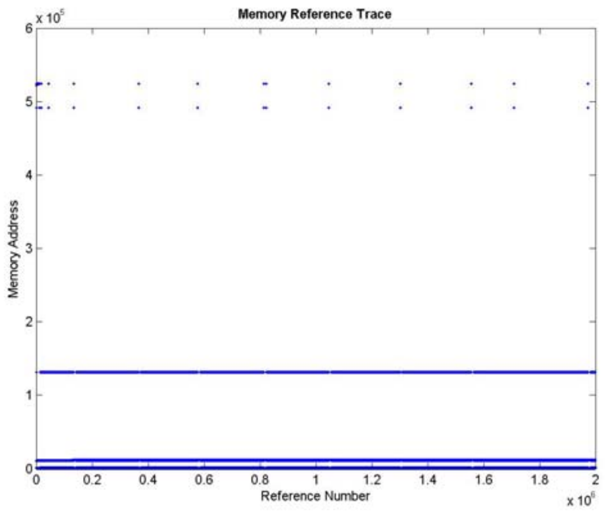
[7] Wikipedia contributors, 'Demand paging', *Wikipedia, The Free Encyclopedia,* 14 August 2006, 14:44 UTC, <http://en.wikipedia.org/w/index.php?title=Demand_paging&oldid=69591315> [accessed 5 October 2006]

Appendix A – Memory Reference Profiles

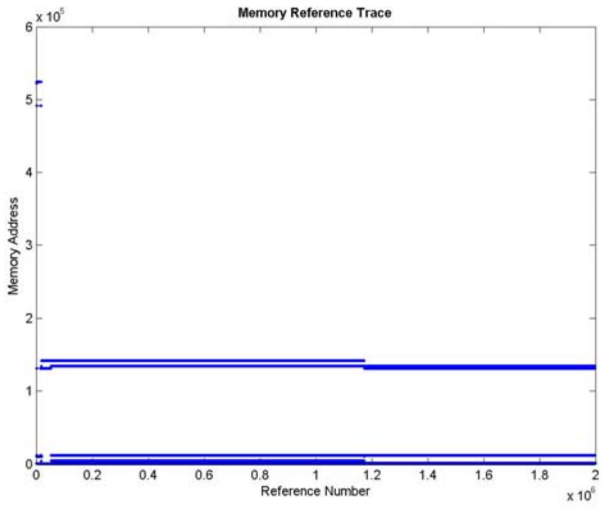
acroread Memory Reference Profile:



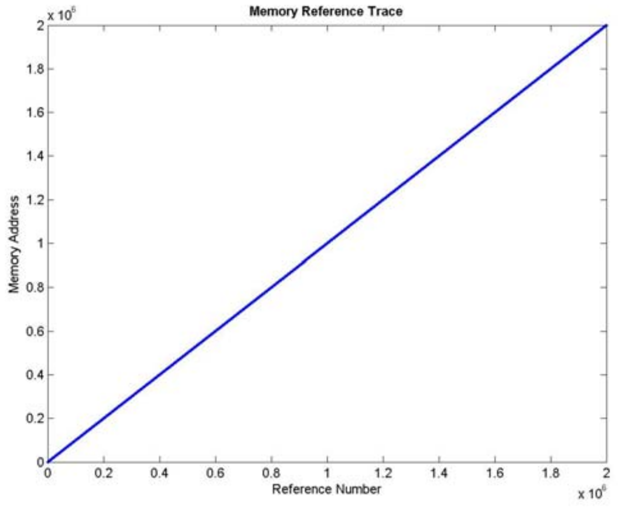
cc1 Memory Reference Profile:



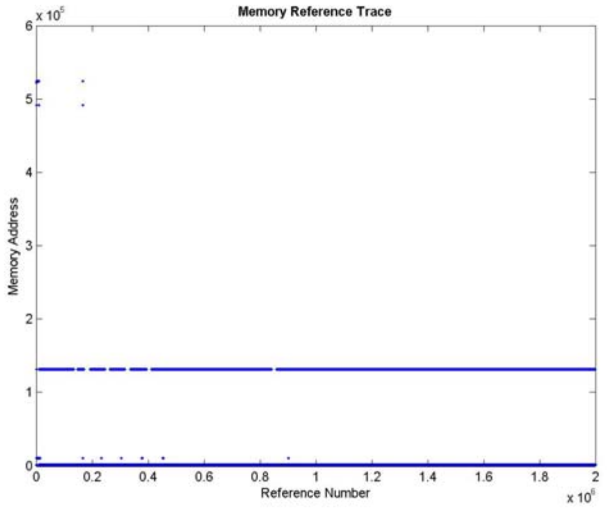
compress95 Memory Reference Profile:



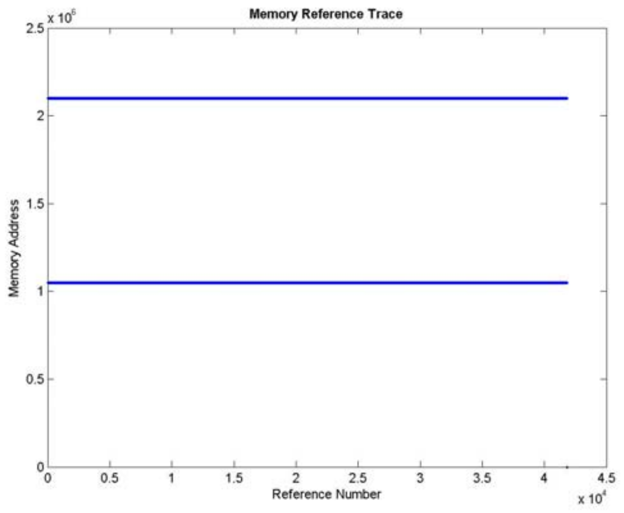
sawtooth Memory Reference Profile:



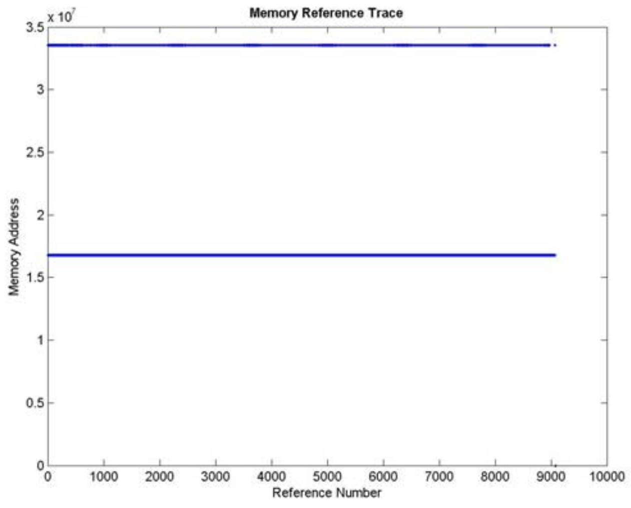
go Memory Reference Profile:



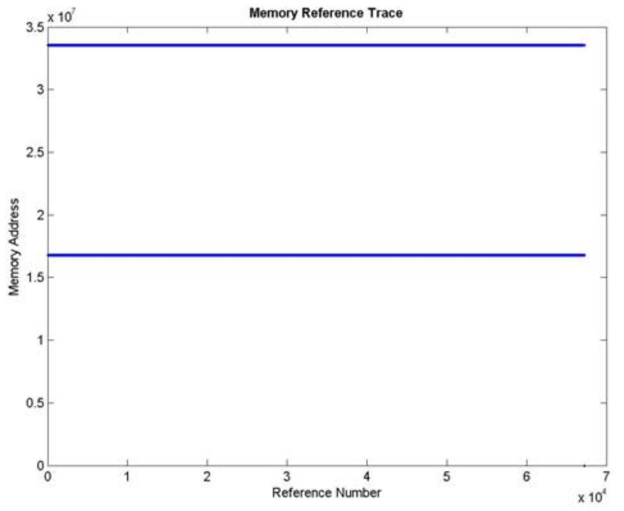
lu Memory Reference Profile:



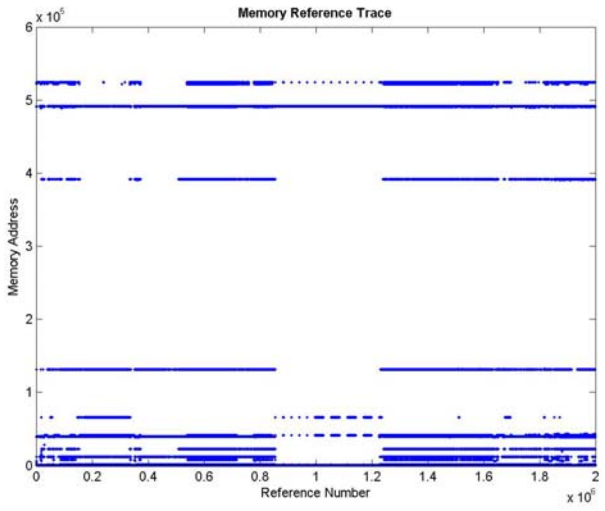
mm16 Memory Reference Profile:



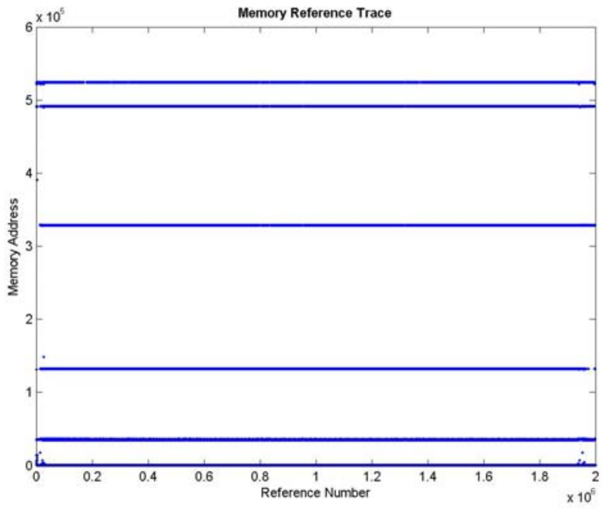
mm32 Memory Reference Profile:



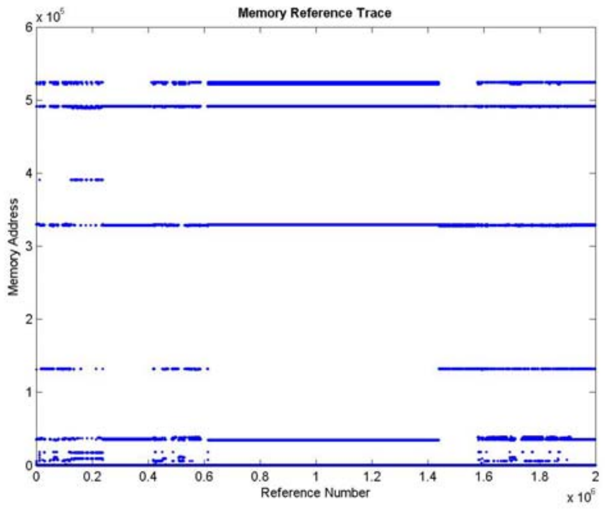
***netscape* Memory Reference Profile:**



***powerpoint* Memory Reference Profile:**



***winword* Memory Reference Profile:**



Appendix B – *DAPsim* Source Code

LRU Page Queue Entry Class: *LRU_pageQ_entry.h*

```
#include <string>
#include <iostream>
#include <fstream>
#include <vector>
#include <queue>
#include <list>
#include <map>
#include <math.h>

#include "main.h"

using namespace std;

class LRU_pageQ_entry {
private:
   unsigned long int pageNumber;
   int lruValue;
   bool resident;
   //int histValue;

public:
   LRU_pageQ_entry();
   LRU_pageQ_entry(unsigned long int newPageNumber, int newLruValue, bool newResidency);

   void setPageNumber(unsigned long int newPageNumber);
   void setLRUValue(int newLruValue);
   void setResidency(bool newResidency);
   //void setHistValue(int newHistValue);

   unsigned long int getPageNumber(void);
   int getLRUValue(void);
   bool getResidency(void);
   //int getHistValue(void);

   //void incHistValue(void);

};
```

LRU Page Queue Entry Source: *LRU_pageQ_entry.c*

```
#include "main.h"
#include "LRU_pageQ_entry.h"

LRU_pageQ_entry::LRU_pageQ_entry()
{
   pageNumber = 0;
   lruValue = 0;
   resident = true;
   //histValue = 0;
}
LRU_pageQ_entry::LRU_pageQ_entry(unsigned long int newPageNumber, int newLruValue, bool
newResidency)
{
   pageNumber = newPageNumber;
   lruValue = newLruValue;
   resident = newResidency;
   //histValue = newHistValue;
}
void LRU_pageQ_entry::setPageNumber(unsigned long int newPageNumber)
{
   pageNumber = newPageNumber;
}
void LRU_pageQ_entry::setLRUValue(int newLruValue)
{
   lruValue = newLruValue;
}
void LRU_pageQ_entry::setResidency(bool newResidency)
{
   resident = newResidency;
}
unsigned long int LRU_pageQ_entry::getPageNumber(void)
{
   return pageNumber;
}
```

```
int LRU_pageQ_entry::getLRUValue(void)
{
   return lruValue;
}
bool LRU_pageQ_entry::getResidency(void)
{
   return resident;
}
```

LRU Page Queue Class: *LRU_pageQ.h*

```
#ifndef LRU_PAGEQ_H_
#define LRU_PAGEQ_H_

#include <string>
#include <iostream>
#include <fstream>
#include <vector>
#include <queue>
#include <list>
#include <map>
#include <math.h>

#include "main.h"
#include "LRU_pageQ_entry.h"

using namespace std;

class LRU_pageQ {
private:
   int size;
   int maxSize;
   list<LRU_pageQ_entry> queue;
   LRU_pageQ_entry LRU_entry;
   LRU_pageQ_entry MRU_entry;

public:
   LRU_pageQ();
   LRU_pageQ(int newMaxSize);

   void setMaxSize(int newMaxSize);

   int getSize(void);
   int getMaxSize(void);
   LRU_pageQ_entry getLRU_entry(void);
   LRU_pageQ_entry getMRU_entry(void);

   void pushMRU(unsigned long int newPage);
   LRU_pageQ_entry popLRU(void);

   //bool referenceUsedPage(long int pageNumber);
   //LRU_pageQ_entry referencePrepagedPage(long int pageNumber);

   void addUsedPage(unsigned long int newPage);
   void addPrepagedPage(unsigned long int newPage);

   void evictPage(void);

   list<LRU_pageQ_entry> * getQueue(void);

   bool pagePresentInQ(unsigned long int pageNumber);
};

#endif
```

LRU Page Queue Source: *LRU_pageQ.c*

```
#include "main.h"
#include "LRU_pageQ.h"

LRU_pageQ::LRU_pageQ()
{
   size = 0;
   maxSize = DEFAULT_MAX_MEM_SIZE;
}
LRU_pageQ::LRU_pageQ(int newMaxSize)
{
   size = 0;
   maxSize = newMaxSize;
}
```

```
void LRU_pageQ::setMaxSize(int newMaxSize)
{
   maxSize = newMaxSize;
}
int LRU_pageQ::getSize(void)
{
   size = queue.size();
   return size;
}
int LRU_pageQ::getMaxSize(void)
{
   return maxSize;
}
LRU_pageQ_entry LRU_pageQ::getLRU_entry(void)
{
   return queue.back();
}
LRU_pageQ_entry LRU_pageQ::getMRU_entry(void)
{
   return queue.front();
}
void LRU_pageQ::pushMRU(unsigned long int newPage)
{
   if ((queue.size() + 1) > maxSize) return;
   LRU_pageQ_entry newEntry;
   newEntry.setPageNumber(newPage);
   newEntry.setLRUValue(0);
   newEntry.setResidency(true);
   //newEntry.setHistValue(0);
   queue.push_front(newEntry);

   // record new queue size
   size = queue.size();
}
LRU_pageQ_entry LRU_pageQ::popLRU(void)
{
   LRU_pageQ_entry LRUentry = queue.back();
   queue.pop_back();

   // record new queue size
   size = queue.size();

   return LRUentry;
}
void LRU_pageQ::evictPage(void)
{
   popLRU();

   // record new queue size
   size = queue.size();
}
list<LRU_pageQ_entry> * LRU_pageQ::getQueue(void)
{
   return &queue;
}
bool LRU_pageQ::pagePresentInQ(unsigned long int pageNumber)
{
   bool pagePresent = false;
   list<LRU_pageQ_entry>::iterator position;
    for (position = queue.begin(); position != queue.end(); ++position) {
        if ((*position).getPageNumber() == pageNumber) {
         pagePresent = true;
         break;
      }
    }

   return pagePresent;
}
```

Used Page Queue Class: *UsedPageQ.h*

```
#include <string>
#include <iostream>
#include <fstream>
#include <vector>
#include <queue>
#include <list>
#include <map>
#include <math.h>

#include "main.h"
```

```
#ifndef LRU_PAGEQ_H_
#include "LRU_pageQ.h"
#endif //LRU_PAGEQ_H_

using namespace std;

class UsedPageQ {
private:
   int size;
   int maxSize;
   int minSize;
   LRU_pageQ ResidentQ;
   LRU_pageQ NonResidentQ;
   int histogram[DEFAULT_MAX_MEM_SIZE];

public:
   UsedPageQ();

   void setMaxSize(int newMaxSize);
   void setMinSize(int newMinSize);

   int getSize(void);
   int getMaxSize(void);
   int getMinSize(void);

   bool referencePage(unsigned long int pageNumber);

   void addPage(unsigned long int newPage);

   void evictPage(void);
   void removeNonResidentPage(unsigned long int pageNumber);

   void printQueueEntries(void);

   void clearHistogram(void);
   void incHistogram(int index);
   int findHistogramIndex(unsigned long int pageNumber);
   void printHistogram(void);
   void decayHistogram(double decayValue);
   int getHistogramEntry(int index);

   bool pagePresentInResidentQ(unsigned long int pageNumber);
   bool pagePresentInNonResidentQ(unsigned long int pageNumber);

   long int getMRUPageNumber(void);
};
```

Used Page Queue Source: *UsedPageQ.c*

```
#include "main.h"
#include "UsedPageQ.h"

int debugModeUsed = DEBUG_MODE;

UsedPageQ::UsedPageQ()
{
   size = 0;
   maxSize = DEFAULT_MAX_MEM_SIZE;
   ResidentQ.setMaxSize(maxSize);
   NonResidentQ.setMaxSize(DEFAULT_MAX_MEM_SIZE);
   clearHistogram();
}

void UsedPageQ::setMaxSize(int newMaxSize)
{
   int currentSize = getSize();
   maxSize = newMaxSize;
   ResidentQ.setMaxSize(maxSize);
   //if (debugModeUsed == DEBUG_4) cout << "Used Queue Max Size set to: " << ResidentQ.getMaxSize()
<< endl;

   // If New Max Size is less than old, remove extra pages
   int elementNum;
   int numOfPagesToRemove = 0;
   if (newMaxSize < currentSize) {
      numOfPagesToRemove = currentSize - newMaxSize;
      for (elementNum = 0; elementNum < numOfPagesToRemove; ++elementNum) {
         NonResidentQ.pushMRU((ResidentQ.popLRU()).getPageNumber());
      }
   }
}

void UsedPageQ::setMinSize(int newMinSize)
{
   minSize = newMinSize;
}

int UsedPageQ::getSize(void)
{
   size = ResidentQ.getSize();
   return size;
}

int UsedPageQ::getMaxSize(void)
{
   return maxSize;
}

int UsedPageQ::getMinSize(void)
{
   return minSize;
}

bool UsedPageQ::referencePage(unsigned long int pageNumber)
{
   bool pageHit = false;   // Default = Page Fault

   int elementNum = 0;
   LRU_pageQ_entry entryReferenced;
   list<LRU_pageQ_entry>::iterator position;
    for (position = ResidentQ.getQueue()->begin(); position != ResidentQ.getQueue()->end();
++position) {
        if ((*position).getPageNumber() == pageNumber) {
         pageHit = true;
         incHistogram(elementNum);
         break;
      }
      elementNum++;
    }
   // If pageNumber not in Resident Queue, mark as Page Fault
   if (!pageHit) {
      // Check Non-Resident Queue for page
      elementNum = maxSize;
      for (position = NonResidentQ.getQueue()->begin(); position != NonResidentQ.getQueue()->end();
++position) {
          if ((*position).getPageNumber() == pageNumber) {
            incHistogram(elementNum);
            removeNonResidentPage((*position).getPageNumber());
            break;
         }
         elementNum++;
      }

      return pageHit;
   }

   entryReferenced = *position;
   ResidentQ.getQueue()->erase(position);
   ResidentQ.getQueue()->push_front(entryReferenced);

   return pageHit;
}

void UsedPageQ::addPage(unsigned long int newPage)
{
   // Check if Eviction is necessary
   if ((ResidentQ.getSize() + 1) > maxSize) {
      //Evict LRU Page
      evictPage();
   }
   ResidentQ.pushMRU(newPage);
   // Update size
   size = ResidentQ.getSize();
}

void UsedPageQ::evictPage(void)
{
   LRU_pageQ_entry evictedPage = ResidentQ.popLRU();
   size = ResidentQ.getSize();
   NonResidentQ.pushMRU(evictedPage.getPageNumber());
}
```

```
void UsedPageQ::removeNonResidentPage(unsigned long int pageNumber)
{
   list<LRU_pageQ_entry>::iterator position;
    for (position = NonResidentQ.getQueue()->begin(); position != NonResidentQ.getQueue()->end();
++position) {
        if ((*position).getPageNumber() == pageNumber) {
         // Remove Page
         NonResidentQ.getQueue()->erase(position);
         return;
      }
    }
}

void UsedPageQ::printQueueEntries(void)
{
   cout << "Used Page Queue:\n";
   cout << "Resident Queue:\n";
   cout << "\tPage #\tHist Value\n";
   cout << "\t------\t----------\n";

   int elementNum = 0;
   list<LRU_pageQ_entry>::iterator position;
    for (position = ResidentQ.getQueue()->begin(); position != ResidentQ.getQueue()->end();
++position) {
      if (elementNum == 0) cout << "    MRU\t";
      else if (elementNum == (ResidentQ.getSize()-1)) cout << "    LRU\t";
      else cout << "\t";
      cout << (*position).getPageNumber() << "\t";
      //cout << (*position).getHistValue() << "\n";
      cout << histogram[elementNum] << "\n";
      elementNum++;
   }

   //elementNum = ResidentQ.getSize() + 1;
   elementNum = maxSize;
   cout << "Non-Resident Queue:\n";
   cout << "\tPage #\tHist Value\n";
   cout << "\t------\t----------\n";
   cout << "    MRU\t";
   for (position = NonResidentQ.getQueue()->begin(); position != NonResidentQ.getQueue()->end();
++position) {
      if (elementNum == (NonResidentQ.getSize()-1)) cout << "    LRU\t";
      else cout << "\t";
      cout << (*position).getPageNumber() << "\t";
      //cout << (*position).getHistValue() << "\n";
      cout << histogram[elementNum] << "\n";
      elementNum++;
   }
}

void UsedPageQ::clearHistogram(void)
{
   int index;
   //int maxIndex = ResidentQ.getMaxSize() + NonResidentQ.getMaxSize();
   int maxIndex = DEFAULT_MAX_MEM_SIZE;
   for (index = 0; index < maxIndex; index++) {
      histogram[index] = 0;
   }
}

void UsedPageQ::incHistogram(int index)
{
   if (index >= DEFAULT_MAX_MEM_SIZE) {
      if (debugModeUsed == DEBUG_2) cout << "ERROR: Histogram index out of bounds!\n";
      return;
   }
   histogram[index] += 1;
}

bool UsedPageQ::pagePresentInResidentQ(unsigned long int pageNumber)
{
   bool pagePresent = false;
   list<LRU_pageQ_entry>::iterator position;
    for (position = ResidentQ.getQueue()->begin(); position != ResidentQ.getQueue()->end();
++position) {
        if ((*position).getPageNumber() == pageNumber) {
         pagePresent = true;
         break;
      }
    }
   return pagePresent;
}

bool UsedPageQ::pagePresentInNonResidentQ(unsigned long int pageNumber)
{
   bool pagePresent = false;
   list<LRU_pageQ_entry>::iterator position;
    for (position = NonResidentQ.getQueue()->begin(); position != NonResidentQ.getQueue()->end();
++position) {
        if ((*position).getPageNumber() == pageNumber) {
         pagePresent = true;
         break;
      }
    }

   return pagePresent;
}

long int UsedPageQ::getMRUPageNumber(void)
{
   return (ResidentQ.getMRU_entry()).getPageNumber();
}

int UsedPageQ::findHistogramIndex(unsigned long int pageNumber)
{
   int index = DEFAULT_MAX_MEM_SIZE - 1;
   list<LRU_pageQ_entry>::iterator position;
   int elementNum = ResidentQ.getMaxSize();
   for (position = NonResidentQ.getQueue()->begin(); position != NonResidentQ.getQueue()->end();
++position) {
      if ((*position).getPageNumber() == pageNumber) {
         index = elementNum;
         return index;
      }
      elementNum++;
   }
   if (debugModeUsed == DEBUG_2) cout << "ERROR: Histogram Index not found!\n";
   return index;
}

void UsedPageQ::decayHistogram(double decayValue)
{
   int index;
   int maxIndex = DEFAULT_MAX_MEM_SIZE;
   for (index = 0; index < maxIndex; index++) {
      histogram[index] = (double)histogram[index] * decayValue;
   }
}

void UsedPageQ::printHistogram(void)
{
   cout << "Used Page Queue Histogram:\n";
   int index;
   int maxIndex = ResidentQ.getMaxSize() + NonResidentQ.getSize();
   for (index = 0; index < maxIndex; index++) {
      cout << histogram[index] << endl;
   }
}

int UsedPageQ::getHistogramEntry(int index)
{
   if (index >= DEFAULT_MAX_MEM_SIZE) {
      if (debugModeUsed >= DEBUG_2) cout << "ERROR: Histogram index out of bounds!\n";
      return -1;
   }
   return histogram[index];
}
```

Prepaged Page Queue Class: *PrepagedPageQ.h*

```
#include <string>
#include <iostream>
#include <fstream>
#include <vector>
#include <queue>
#include <list>
#include <map>
#include <math.h>
```

```
#include "main.h"
#ifndef LRU_PAGEQ_H_
#include "LRU_pageQ.h"
#endif //LRU_PAGEQ_H_

using namespace std;

class PrepagedPageQ {
private:
   int consumedAlloc;
   int targetAlloc;
   LRU_pageQ ResidentQ;
   LRU_pageQ NonResidentQ;
   LRU_pageQ DegreeQ;
   int histogram[DEFAULT_MAX_MEM_SIZE];
   int degreeHistogram[DEFAULT_MAX_DEGREE + 2];

public:
   PrepagedPageQ();
   PrepagedPageQ(int newTargetAlloc);

   void setTargetAlloc(int newTargetAlloc);

   int getConsumedAlloc(void);
   int getTargetAlloc(void);

   LRU_pageQ_entry referencePage(unsigned long int pageNumber);

   void addPage(unsigned long int newPage);

   void evictPage(void);

   void printQueueEntries(void);
   void printResidentQueueEntries(void);

   void clearHistogram(void);
   void clearDegreeHistogram(void);
   void incHistogram(int index);
   void decayHistogram(double decayValue);
   void printHistogram(void);
   int getHistogramEntry(int index);

   bool pagePresentInQ(unsigned long int pageNumber);

   void removeResidentPage(unsigned long int pageNumber);
   void removeNonResidentPage(unsigned long int pageNumber);

   void updateDegreeHistogram(unsigned long int pageNumber, int currentDegree);
   void addDegreePage(unsigned long int newPage);
   void incDegreeHistogram(int index);
   int getDegreeHistogramEntry(int index);
   void printDegreeQueue(int currentDegree);
   int getDegreeQueueSize(void);
};
```

Prepaged Page Queue Source: *PrepagedPageQ.c*

```
#include "main.h"
#include "PrepagedPageQ.h"

int debugModePrepaged = DEBUG_MODE;

PrepagedPageQ::PrepagedPageQ()
{
   // Set Prepaged Target Allocation
   targetAlloc = DEFAULT_PREPAGED_TARGET_ALLOC;
   ResidentQ.setMaxSize(targetAlloc);

   // Set Prepaged Consumed Allocation
   consumedAlloc = 0;

   clearHistogram();

   DegreeQ.setMaxSize(DEFAULT_MAX_DEGREE+2);
}

PrepagedPageQ::PrepagedPageQ(int newTargetAlloc)
{
   // Set Prepaged Target Allocation
   targetAlloc = newTargetAlloc;
   ResidentQ.setMaxSize(targetAlloc);
   // Set Prepaged Consumed Allocation
   consumedAlloc = 0;

   clearHistogram();

   DegreeQ.setMaxSize(DEFAULT_MAX_DEGREE+2);
}

void PrepagedPageQ::setTargetAlloc(int newTargetAlloc)
{
   int oldConsumedAlloc = getConsumedAlloc();

   if (newTargetAlloc > DEFAULT_MAX_PREPAGED_TARGET_ALLOC) targetAlloc =
DEFAULT_MAX_PREPAGED_TARGET_ALLOC;
   else if (newTargetAlloc < DEFAULT_MIN_PREPAGED_TARGET_ALLOC) targetAlloc =
DEFAULT_MIN_PREPAGED_TARGET_ALLOC;
   else targetAlloc = newTargetAlloc;
   ResidentQ.setMaxSize(targetAlloc);

   // If New Target Allocation is less than old consumed allocation, place extra pages into non-
resident queue
   int elementNum;
   int numOfPagesToRemove = 0;
   if (newTargetAlloc < oldConsumedAlloc) {
      numOfPagesToRemove = oldConsumedAlloc - newTargetAlloc;
      for (elementNum = 0; elementNum < numOfPagesToRemove; ++elementNum) {
         NonResidentQ.pushMRU(ResidentQ.popLRU().getPageNumber());
      }
   }

   // Reset Consumed Allocation
   consumedAlloc = ResidentQ.getSize();
}

int PrepagedPageQ::getConsumedAlloc(void)
{
   // Refresh Consumed Allocation
   consumedAlloc = ResidentQ.getSize();

   return consumedAlloc;
}

int PrepagedPageQ::getTargetAlloc(void)
{
   return targetAlloc;
}

LRU_pageQ_entry PrepagedPageQ::referencePage(unsigned long int pageNumber)
{
   LRU_pageQ_entry referencedPage;
   bool pageHit = false;

   // Check Resident Queue
   int elementNum = 0;
   list<LRU_pageQ_entry>::iterator position;
   for (position = ResidentQ.getQueue()->begin(); position != ResidentQ.getQueue()->end();
++position) {
      if ((*position).getPageNumber() == pageNumber) {
         pageHit = true;
         incHistogram(elementNum);
         referencedPage = *position;
         removeResidentPage(pageNumber);
         return referencedPage;
      }
      else elementNum++;
    }

   // Check Non-Resident Queue (for purposes of updating histogram)
   elementNum = targetAlloc;
   for (position = NonResidentQ.getQueue()->begin(); position != NonResidentQ.getQueue()->end();
++position) {
      if (elementNum >= DEFAULT_MAX_MEM_SIZE) break;
      if ((*position).getPageNumber() == pageNumber) {
         //cout << "Incrementing Prepage Histogram at " << elementNum << " for Non-Resident Page "
<< (*position).getPageNumber() << endl;
         incHistogram(elementNum);
         break;
      }
      else elementNum++;
    }
   // No page hit
   referencedPage.setPageNumber(NULL);

   return referencedPage;
}

void PrepagedPageQ::addPage(unsigned long int newPage)
{
   // Check if already present in prepage queue
   if (pagePresentInQ(newPage)) {
      removeResidentPage(newPage);
   }

   // Check if Eviction is necessary
   if ((ResidentQ.getSize() + 1) > targetAlloc) {  //Evict LRU Page
      evictPage();
   }
   ResidentQ.pushMRU(newPage);
   // Update Consumed Allocation
   consumedAlloc = ResidentQ.getSize();
}

void PrepagedPageQ::evictPage(void)
{
   // Evict Page
   // Add Evicted Page to NonResident Queue
   NonResidentQ.pushMRU(ResidentQ.popLRU().getPageNumber());

   // Update Consumed Allocation
   consumedAlloc = ResidentQ.getSize();
}

void PrepagedPageQ::removeResidentPage(unsigned long int pageNumber)
{
   list<LRU_pageQ_entry>::iterator position;
    for (position = ResidentQ.getQueue()->begin(); position != ResidentQ.getQueue()->end();
++position) {
        if ((*position).getPageNumber() == pageNumber) {
         // Remove Page
         ResidentQ.getQueue()->erase(position);
         // Update Consumed Alloc
         consumedAlloc = ResidentQ.getSize();
         return;
      }
    }
}

void PrepagedPageQ::removeNonResidentPage(unsigned long int pageNumber)
{
   list<LRU_pageQ_entry>::iterator position;
    for (position = NonResidentQ.getQueue()->begin(); position != NonResidentQ.getQueue()->end();
++position) {
        if ((*position).getPageNumber() == pageNumber) {
         // Remove Page
         NonResidentQ.getQueue()->erase(position);
         return;
      }
    }
}

void PrepagedPageQ::printQueueEntries(void)
{
   cout << "Prepaged Page Queue:\n";
   cout << "Resident Queue:\n";
   cout << "Entry\tPage #\tHist Value\n";
   cout << "-----\t------\t----------\n";

   // Resident Queue Entries
   int elementNum = 0;
   list<LRU_pageQ_entry>::iterator position;
    for (position = ResidentQ.getQueue()->begin(); position != ResidentQ.getQueue()->end();
++position) {
      if (elementNum == 0) cout << elementNum + 1 << " MRU\t";
      else if (elementNum == (ResidentQ.getSize()-1)) cout << elementNum + 1 << " LRU\t";
      else cout << elementNum + 1 << "\t";
      cout << (*position).getPageNumber() << "\t";
      //cout << (*position).getHistValue() << "\n";
      cout << histogram[elementNum] << "\n";
      elementNum++;
   }
   // NonResident Queue Entries
   elementNum = targetAlloc;
   cout << "Non-Resident Queue:\n";
   cout << "Entry\tPage #\tHist Value\n";
   cout << "-----\t------\t----------\n";
   for (position = NonResidentQ.getQueue()->begin(); position != NonResidentQ.getQueue()->end();
++position) {
      if (elementNum == targetAlloc) cout << elementNum + 1 << " MRU\t";
      else if (elementNum == (targetAlloc + NonResidentQ.getSize() - 1)) cout << elementNum + 1 << "
LRU\t";
      else cout << elementNum + 1 << "\t";
      cout << (*position).getPageNumber() << "\t";
      //cout << (*position).getHistValue() << "\n";
      if (elementNum < DEFAULT_MAX_MEM_SIZE) cout << histogram[elementNum] << "\n";
      else break;
      elementNum++;
   }
}

void PrepagedPageQ::printResidentQueueEntries(void)
{
   cout << "Prepaged Page Queue:\n";
   cout << "Resident Queue:\n";
   cout << "Entry\tPage #\tHist Value\n";
   cout << "-----\t------\t----------\n";

   int elementNum = 0;
   list<LRU_pageQ_entry>::iterator position;
    for (position = ResidentQ.getQueue()->begin(); position != ResidentQ.getQueue()->end();
++position) {
      if (elementNum == 0) cout << elementNum + 1 << " MRU\t";
      else if (elementNum == (ResidentQ.getSize()-1)) cout << elementNum + 1 << " LRU\t";
      else cout << elementNum + 1 << "\t";
      cout << (*position).getPageNumber() << "\t";
      //cout << (*position).getHistValue() << "\n";
      cout << histogram[elementNum] << "\n";
      elementNum++;
   }
}

void PrepagedPageQ::clearHistogram(void)
{
   int index;
   for (index = 0; index < DEFAULT_MAX_MEM_SIZE; index++) {
      histogram[index] = 0;
   }
}

void PrepagedPageQ::clearDegreeHistogram(void)
{
   int index;
   for (index = 0; index < (DEFAULT_MAX_DEGREE + 2); index++) {
      degreeHistogram[index] = 0;
   }
}

void PrepagedPageQ::incHistogram(int index)
{
   if (index >= DEFAULT_MAX_MEM_SIZE) {
      if (debugModePrepaged == DEBUG_2) cout << "ERROR: Histogram index out of bounds!\n";
      return;
   }

   histogram[index] += 1;
}

void PrepagedPageQ::incDegreeHistogram(int index)
{
   if (index > (DEFAULT_MAX_DEGREE + 1)) {
      if (debugModePrepaged == DEBUG_2) cout << "ERROR: Degree histogram index out of bounds!\n";
      return;
   }

   degreeHistogram[index] += 1;
}

int PrepagedPageQ::getHistogramEntry(int index)
{
   if (index >= DEFAULT_MAX_MEM_SIZE) {
      if (debugModePrepaged == DEBUG_2) cout << "ERROR: Histogram index out of bounds!\n";
      return -1;
   }

   return histogram[index];
}

bool PrepagedPageQ::pagePresentInQ(unsigned long int pageNumber)
{
   bool pagePresent = false;
   list<LRU_pageQ_entry>::iterator position;
    for (position = ResidentQ.getQueue()->begin(); position != ResidentQ.getQueue()->end();
++position) {
        if ((*position).getPageNumber() == pageNumber) {
         pagePresent = true;
         break;
      }
    }

   return pagePresent;
}

void PrepagedPageQ::decayHistogram(double decayValue)
{
   int index;
   for (index = 0; index < DEFAULT_MAX_MEM_SIZE; index++) {
      histogram[index] = (double)histogram[index] * decayValue;   //Truncation allowed
   }
}

void PrepagedPageQ::printHistogram(void)
{
   cout << "Prepaged Page Queue Histogram:\n";
   int index;
   for (index = 0; index < targetAlloc; index++) {
      cout << histogram[index] << endl;
   }
}

void PrepagedPageQ::printDegreeQueue(int currentDegree)
{
   cout << "Prepaged Degree Queue:\n";
   cout << "\tEntry\tHist\n";
   cout << "\t-----\t----\n";

   int index = 0;
   list<LRU_pageQ_entry>::iterator position;
    for (position = DegreeQ.getQueue()->begin(); position != DegreeQ.getQueue()->end(); ++position)
{
      if (index > (currentDegree + 1)) break;
      if (index == 0) cout << "MRU\t";
      else cout << "\t";
      cout << (*position).getPageNumber() << "\t" << degreeHistogram[index] << endl;
      index++;
   }
}

void PrepagedPageQ::updateDegreeHistogram(unsigned long int pageNumber, int currentDegree)
{
   // Check Degree Queue
   int elementNum = 0;
   list<LRU_pageQ_entry>::iterator position;
   for (position = DegreeQ.getQueue()->begin(); position != DegreeQ.getQueue()->end(); ++position) {
      if (elementNum > (currentDegree + 1)) break;
      else if ((*position).getPageNumber() == pageNumber) {
         incDegreeHistogram(elementNum);
      }
      elementNum++;
    }
}

void PrepagedPageQ::addDegreePage(unsigned long int newPage)
{
   if ((DegreeQ.getSize() + 1) > DegreeQ.getMaxSize()) { //Evict LRU Page
      DegreeQ.evictPage();
   }

   DegreeQ.pushMRU(newPage);
}

int PrepagedPageQ::getDegreeHistogramEntry(int index)
{
   if (index > (DEFAULT_MAX_DEGREE + 1)) {
      if (debugModePrepaged == DEBUG_2) cout << "ERROR: Degree histogram index out of bounds!\n";
      return 0;
   }

   return degreeHistogram[index];
}

int PrepagedPageQ::getDegreeQueueSize(void)
{
   return DegreeQ.getSize();
}
```

Virtual Memory Management Unit Class: *VirtualMMU.h*

```
#include <string>
#include <iostream>
#include <fstream>
#include <vector>
#include <queue>
#include <list>
#include <map>
#include <math.h>

#include "main.h"
#include "UsedPageQ.h"
#include "PrepagedPageQ.h"

using namespace std;

class VirtualMMU {
private:
   int MMSize;       // Size of Main Memory
   int prepagedAlloc;   // Max size of Prepaged Queue
   int degree;
   int predictionScheme;
   int historySize;
   UsedPageQ usedPageQ;
   PrepagedPageQ prepagedPageQ;

public:
   // Prediction lists used to determine most effective prediction scheme
   vector<unsigned long int> addressLocalPredictions;
   vector<unsigned long int> recencyLocalPredictions;
   vector<unsigned long int> stridePredictions;
   vector<unsigned long int> hybridPredictions;
   // Prediction histograms
   int addressLocalHistogram[DEFAULT_MAX_DEGREE];
   int recencyLocalHistogram[DEFAULT_MAX_DEGREE];
   int strideHistogram[DEFAULT_MAX_DEGREE];
   int hybridHistogram[DEFAULT_MAX_DEGREE];

   // Global Variables
   long int transfers;
   double avg_transfer;
   double transfer_rate;

   VirtualMMU();
   VirtualMMU(int newMMSize, int newPrepagedAlloc, int newDegree, int newPredictionScheme);

   void printVMMU(void);

   void setMMSize(int newMMSize);
   void setPrepagedAlloc(int newPrepagedAlloc);
   void setDegree(int newDegree);
   void setHistorySize(int newHistorySize);

   int getMMSize(void);
   int getPrepagedAlloc(void);
   int getDegree(void);
   int getHistorySize(void);

   int incDegree(void);
   int decDegree(void);

   UsedPageQ * getUsedPageQ(void);
   PrepagedPageQ * getPrepagedPageQ(void);

   int handlePageRef(unsigned long int referencedPage, int preParamMode);
   vector<unsigned long int> generatePrepageList(unsigned long int referencedPage);
   void printPrepageList(vector<unsigned long int> *prepageList);
   void fetchPrepages(vector<unsigned long int> prepageList);

   void updatePrepagedAlloc(void);
   void updateDegree(void);
   void updatePredictionScheme(void);

   unsigned int calculateBenefit(int targetAlloc);
   unsigned int calculateCost(int targetAlloc);

   void initializePredictionHistograms(void);
   void clearPredictionHistograms(void);
   int calculateHistTotal(int histogram[]);
   void updatePredictionHistograms(unsigned long int referencedPage);
   void printPredictionHistograms(void);
   void printPredictionLists(void);

};
```

Virtual Memory Management Unit Source: *VirtualMMU.c*

```
#include "main.h"
#include "VirtualMMU.h"

string predictionSchemes[] = {"Uninitialized","Address Local","Recency Local","Stride","Hybrid"};

// Global Variables
int debugMode = DEBUG_MODE;
int refNumber = 1;   // Cummulative Reference Number
int faultNumber = 1;// Cummulative Page Fault Number
int decayPeriod = DEFAULT_DECAY_PERIOD;               // Period at which histograms are decayed
int allocUpdatePeriod = DEFAULT_ALLOC_UPDATE_PERIOD;  // Period at which the prepaged target
allocation is updated
double decayValue = (double)DEFAULT_DECAY_NUMERATOR/(double)DEFAULT_DECAY_DENOMINATOR;

VirtualMMU::VirtualMMU()
{
   MMSize = DEFAULT_MEM_SIZE;
   prepagedAlloc = DEFAULT_PREPAGED_TARGET_ALLOC;
   degree = DEFAULT_DEGREE;
   predictionScheme = DEFAULT_PREDICTION_SCHEME;

   usedPageQ.setMaxSize(MMSize - prepagedPageQ.getConsumedAlloc());
   usedPageQ.setMinSize(DEFAULT_MIN_USED_Q_SIZE);
   prepagedPageQ.setTargetAlloc(prepagedAlloc);

   initializePredictionHistograms();

   refNumber = 1; // Cummulative Reference Number
   faultNumber = 1;// Cummulative Page Fault Number
}

VirtualMMU::VirtualMMU(int newMMSize, int newPrepagedAlloc, int newDegree, int newPredictionScheme)
{
   MMSize = newMMSize;
   prepagedAlloc = newPrepagedAlloc;
   degree = newDegree;
   predictionScheme = newPredictionScheme;

   usedPageQ.setMaxSize(MMSize - prepagedPageQ.getConsumedAlloc());
   usedPageQ.setMinSize(DEFAULT_MIN_USED_Q_SIZE);
   prepagedPageQ.setTargetAlloc(prepagedAlloc);

   initializePredictionHistograms();

   refNumber = 1; // Cummulative Reference Number
   faultNumber = 1;// Cummulative Page Fault Number
}
void VirtualMMU::printVMMU()
{
   cout << "VMMU" << endl;
   cout << "--------" << endl;
   cout << "Main Memory Size: " << MMSize << endl;
   cout << "Prepaged Allocation: " << prepagedAlloc << endl;
   cout << "Degree: " << degree << endl;
   cout << "Prediciton Scheme: " << predictionScheme << endl;
   cout << "History Size: " << historySize << endl;
   usedPageQ.printQueueEntries();
   cout << endl << endl;
   prepagedPageQ.printQueueEntries();
}
void VirtualMMU::setMMSize(int newMMSize)
{
   MMSize = newMMSize;

   usedPageQ.setMaxSize(MMSize - prepagedPageQ.getConsumedAlloc());
}
void VirtualMMU::setPrepagedAlloc(int newPrepagedAlloc)
{
   if (newPrepagedAlloc > DEFAULT_MAX_PREPAGED_TARGET_ALLOC) {
      prepagedAlloc = DEFAULT_MAX_PREPAGED_TARGET_ALLOC;
   }
   else if (newPrepagedAlloc < DEFAULT_MIN_PREPAGED_TARGET_ALLOC) {
      prepagedAlloc = DEFAULT_MIN_PREPAGED_TARGET_ALLOC;
   }
   else {
      prepagedAlloc = newPrepagedAlloc;
   }

   usedPageQ.setMaxSize(MMSize - prepagedPageQ.getConsumedAlloc());
   prepagedPageQ.setTargetAlloc(prepagedAlloc);
}
void VirtualMMU::setDegree(int newDegree)
{
   degree = newDegree;
}
void VirtualMMU::setHistorySize(int newHistorySize)
{
   historySize = newHistorySize;
}
int VirtualMMU::getMMSize(void)
{
   return MMSize;
}
int VirtualMMU::getPrepagedAlloc(void)
{
   return prepagedAlloc;
}
int VirtualMMU::getDegree(void)
{
   return degree;
}
int VirtualMMU::incDegree(void)
{
   int newDegree = degree + 1;
   if (newDegree > DEFAULT_MAX_DEGREE) newDegree = DEFAULT_MAX_DEGREE;
   return newDegree;
}
int VirtualMMU::decDegree(void)
{
   int newDegree = degree - 1;
   if (newDegree < DEFAULT_MIN_DEGREE) newDegree = DEFAULT_MIN_DEGREE;
   return newDegree;
}
int VirtualMMU::getHistorySize(void)
{
   return historySize;
}

UsedPageQ * VirtualMMU::getUsedPageQ(void)
{
   return &usedPageQ;
}

PrepagedPageQ * VirtualMMU::getPrepagedPageQ(void)
{
   return &prepagedPageQ;
}

int VirtualMMU::handlePageRef(unsigned long int referencedPage, int preParamMode)
{
   if (debugMode == DEBUG_2) {
      cout << "----------------------------\n";
      cout << "Referencing Page #" << referencedPage << endl;
      cout << "----------------------------\n";
   }

   int refResult = PAGE_FAULT;
   LRU_pageQ_entry prepagedPage;
   vector<unsigned long int> prepageList;
   int prepageParameterMode = preParamMode;

   // Check if page fault //
   // Check Used Page Queue
   if (usedPageQ.referencePage(referencedPage)) {
      // Page hit in Used Page Queue, no further action required
      if (debugMode == DEBUG_2) cout << "Page hit in Used Page Queue\n";
      refResult = PAGE_HIT;
   }
   else { // Check Prepaged Page Queue
      prepagedPageQ.updateDegreeHistogram(referencedPage, degree);
      updatePredictionHistograms(referencedPage);
      prepagedPage = prepagedPageQ.referencePage(referencedPage);
      usedPageQ.setMaxSize(MMSize - prepagedPageQ.getConsumedAlloc());  //Update Used Page Q MaxSize
      // Check if page fault
      if (prepagedPage.getPageNumber() == NULL) {  // Page Fault in Prepaged Queue
         // PAGE FAULT //
         refResult = PAGE_FAULT;

         // Generate Prepage List
         prepageList = generatePrepageList(referencedPage);

         // Add referenced page to Used Page Queue
         usedPageQ.addPage(referencedPage);
         ++transfers;

         // Fetch Prepaged Pages
         fetchPrepages(prepageList);

         // Update Prepage Parameters
         if (prepageParameterMode == DYNAMIC_PREPAGE_PARAMETERS || prepageParameterMode ==
DYNAMIC_PREPAGE_ALLOCATION) {
            // Update Target Allocation
            if ((faultNumber % allocUpdatePeriod) == 0) {
               if (refNumber > DEFAULT_ALLOC_STARTUP_PERIOD) updatePrepagedAlloc();
            }
         }

         if (prepageParameterMode == DYNAMIC_PREPAGE_PARAMETERS || prepageParameterMode ==
DYNAMIC_PREDICTION_METHOD) {
            // Update Prediction Scheme
            updatePredictionScheme();
         }

         if (prepageParameterMode == DYNAMIC_PREPAGE_PARAMETERS || prepageParameterMode ==
DYNAMIC_DEGREE) {
            // Update Degree
            if (debugMode == DEBUG_5) prepagedPageQ.printDegreeQueue(degree);
            updateDegree();
         }

         ++faultNumber;
      }
      else {                              // Push referenced page into Used Queue
         if (debugMode == DEBUG_2) {
            cout << "Page hit in Prepaged Page Queue\n";
            cout << "Transferring page from Prepaged Page Queue to Used Page Queue\n";
         }

         refResult = PAGE_HIT;
         usedPageQ.addPage(prepagedPage.getPageNumber());
      }
   }

   if (debugMode == DEBUG_2) {
      usedPageQ.printQueueEntries();
      cout << "\n\n";
      prepagedPageQ.printQueueEntries();
      cout << "\n\n";
   }

   // Periodically Decay Histogram
   if ((refNumber % decayPeriod) == 0) {
      // Perform Decay
      prepagedPageQ.decayHistogram(decayValue);
      usedPageQ.decayHistogram(decayValue);
   }
   refNumber++;
   return refResult;
}

vector<unsigned long int> VirtualMMU::generatePrepageList(unsigned long int referencedPage)
{
   if (debugMode == DEBUG_2) {
      cout << "Generating Prepage List using " << predictionSchemes[predictionScheme] << "
prediciton ... \n";
      cout << "Referenced page = " << referencedPage << endl;
   }
   vector<unsigned long int> prepageList;
   vector<unsigned long int> degreeList;
   int predictionNumber;
   unsigned long int currentPrediction;

   // Address Local Prediction //
   vector<unsigned long int> addressLocalPredictionList;
   vector<unsigned long int> addressLocalDegreeList;
   for (predictionNumber = 1; predictionNumber <= (degree+2); predictionNumber++) {
      // Make predicitons
      if (predictionNumber == 1) currentPrediction = referencedPage + 1;
      else {
         if ((predictionNumber % 2) == 0) {  //If Even Prediction Number
            currentPrediction = currentPrediction - predictionNumber;
         }
         else {                        //If Odd Prediction Number
            currentPrediction = currentPrediction + predictionNumber;
         }
      }
      //cout << "Adding prediction: " << currentPrediction << endl;
      if (predictionNumber <= degree) addressLocalPredictionList.push_back(currentPrediction);
      addressLocalDegreeList.push_back(currentPrediction);
   }
   addressLocalPredictions = addressLocalPredictionList;

   // Recency Local Prediction //
   vector<unsigned long int> recencyLocalPredictionList;
   vector<unsigned long int> recencyLocalDegreeList;
   for (predictionNumber = 1; predictionNumber <= (degree+2); predictionNumber++) {
      if (predictionNumber <= degree) recencyLocalPredictionList.push_back(0);
      recencyLocalDegreeList.push_back(0);
   }
   recencyLocalPredictions = recencyLocalPredictionList;

   // Stride Prediction //
   vector<unsigned long int> stridePredictionList;
   vector<unsigned long int> strideDegreeList;
   long int stride;
   //calcualate stride
   stride = referencedPage - usedPageQ.getMRUPageNumber();
   //make predictions
   currentPrediction = referencedPage;
   for (predictionNumber = 1; predictionNumber <= (degree+2); predictionNumber++) {
      // Make predicitons
      currentPrediction += stride;
      if (predictionNumber <= degree) stridePredictionList.push_back(currentPrediction);
      strideDegreeList.push_back(currentPrediction);
   }
   stridePredictions = stridePredictionList;

   // Hybrid Prediction //
   vector<unsigned long int> hybridPredictionList;
   vector<unsigned long int> hybridDegreeList;
   for (predictionNumber = 1; predictionNumber <= (degree+2); predictionNumber++) {
      if (predictionNumber <= degree) hybridPredictionList.push_back(0);
      hybridDegreeList.push_back(0);
   }
   hybridPredictions = hybridPredictionList;

   // Set Page Prediction //
   switch(predictionScheme) {
      case PREDICTION_UNINITIALIZED:
         break;
      case PREDICTION_ADDRESS_LOCAL:
         prepageList = addressLocalPredictionList;
         degreeList = addressLocalDegreeList;
         break;
      case PREDICTION_RECENCY_LOCAL:
         prepageList = recencyLocalPredictionList;
         degreeList = recencyLocalDegreeList;
         break;
      case PREDICTION_STRIDE:
         prepageList = stridePredictionList;
         degreeList = strideDegreeList;
         break;
      case PREDICTION_HYBRID:
         prepageList = hybridPredictionList;
         degreeList = hybridDegreeList;
         break;
      default:
         ;
   }

   // Add degreeList to Degree Queue
   vector<unsigned long int>::iterator predictionNum;
   predictionNum = degreeList.end();
   --predictionNum;
   int predNumber;
    for (predNumber = (degree+2); predNumber > 0; --predNumber) {
      //cout << "Adding " << *predictionNum << " to degree list\n";
      prepagedPageQ.addDegreePage(*predictionNum);

      if (predictionNum == degreeList.begin()) {
         break;
      }
      else {
         --predictionNum;
      }
   }

   // Check if predicted pages are already in main memory
    for (predictionNum = prepageList.begin(); predictionNum != prepageList.end(); ++predictionNum) {
        if (usedPageQ.pagePresentInResidentQ(*predictionNum) ||
prepagedPageQ.pagePresentInQ(*predictionNum)) {
         // Page is present in Main Mem, remove from prepageList
         if (debugMode == DEBUG_2) cout << "Erasing prediction: " << *predictionNum << endl;
         prepageList.erase(predictionNum);
         --predictionNum;
      }
   }

   printPrepageList(&prepageList);
   return prepageList;
}
void VirtualMMU::printPrepageList(vector<unsigned long int> *prepageList)
{
   if (debugMode == DEBUG_2) {
      cout << "Prepage List:\n";
      cout << "Prediciton List size = " << (*prepageList).size() << endl;
   }
   vector<unsigned long int>::iterator predictionNum;
   int predictionNumber = 1;
    for (predictionNum = (*prepageList).begin(); predictionNum != (*prepageList).end();
++predictionNum) {
        if (debugMode == DEBUG_2) cout << predictionNumber << ": " << *predictionNum << endl;
      predictionNumber++;
   }

   if (debugMode == DEBUG_2) cout << "Front = " << (*prepageList).front() << endl;
}
void VirtualMMU::fetchPrepages(vector<unsigned long int> prepageList)
{
   vector<unsigned long int>::iterator predictionNum;
   predictionNum = prepageList.end();
   --predictionNum;
   int predNumber;
    for (predNumber = degree; predNumber > 0; --predNumber) {

      // Add page to Prepaged Page Q
      prepagedPageQ.addPage(*predictionNum);
      ++transfers;

      // Remove page from Non-Resident Queue if present
      prepagedPageQ.removeNonResidentPage(*predictionNum);

      if (predictionNum == prepageList.begin()) {
         break;
      }
      else {
         --predictionNum;
      }
   }
   // Update Used Page Queue Max Size
   usedPageQ.setMaxSize(MMSize - prepagedPageQ.getConsumedAlloc());
}
void VirtualMMU::updatePrepagedAlloc(void)
{
   int newPrepagedAlloc = getPrepagedAlloc();
   int tempPrepagedAlloc;
   int currentNetReduction = 0;
   int maxNetReduction = 0;
   int currentPrepagedAlloc = getPrepagedAlloc();
   if (debugMode == DEBUG_1) cout << "Old Prepaged Allocation: " << prepagedAlloc << endl;

   // Determine new prepage allocation
   unsigned int benefit = 0;
   unsigned int cost = 0;
   for (tempPrepagedAlloc = 0; tempPrepagedAlloc < (MMSize - usedPageQ.getMinSize());
++tempPrepagedAlloc) {
      currentNetReduction = calculateBenefit(tempPrepagedAlloc) - calculateCost(tempPrepagedAlloc);
      if (currentNetReduction > maxNetReduction) {
         maxNetReduction = currentNetReduction;
         newPrepagedAlloc = tempPrepagedAlloc;
      }
   }

   setPrepagedAlloc(newPrepagedAlloc);
   if (debugMode == DEBUG_4) cout << "Setting prepaged allocation to: " << prepagedAlloc << endl;
   //if (debugMode == DEBUG_4) {
   // usedPageQ.printHistogram();
   // prepagedPageQ.printHistogram();
   //}
   if (debugMode == DEBUG_1) cout << "New Prepaged Allocation: " << prepagedAlloc << endl;
}
void VirtualMMU::updatePredictionScheme(void)
{
   int tempPredictionScheme = predictionScheme;
   if (debugMode == DEBUG_1) cout << "Old Prediction Scheme: " <<
predictionSchemes[predictionScheme] << endl;

   // Determine new prediction scheme
   int mostEffectivePrediciton = predictionScheme;
   int maxTotal = 0;
   int currentTotal = 0;
   currentTotal = calculateHistTotal(addressLocalHistogram);
   if (currentTotal > maxTotal) {
      maxTotal = currentTotal;
      mostEffectivePrediciton = PREDICTION_ADDRESS_LOCAL;
   }
   currentTotal = calculateHistTotal(recencyLocalHistogram);
   if (currentTotal > maxTotal) {
      maxTotal = currentTotal;
      mostEffectivePrediciton = PREDICTION_RECENCY_LOCAL;
   }
   currentTotal = calculateHistTotal(strideHistogram);
   if (currentTotal > maxTotal) {
      maxTotal = currentTotal;
      mostEffectivePrediciton = PREDICTION_STRIDE;
   }
   currentTotal = calculateHistTotal(hybridHistogram);
   if (currentTotal > maxTotal) {
      maxTotal = currentTotal;
      mostEffectivePrediciton = PREDICTION_HYBRID;
   }

   predictionScheme = mostEffectivePrediciton;

   if (predictionScheme != tempPredictionScheme){
      if (debugMode == DEBUG_8) {
         cout << "Old Prediction Scheme: " << predictionSchemes[tempPredictionScheme] << endl;
         cout << "New Prediction Scheme: " << predictionSchemes[predictionScheme] << endl;

         printPredictionHistograms();
         //printPredictionLists();
      }
   }

   clearPredictionHistograms();
}
void VirtualMMU::updateDegree(void)
{
   int oldDegree = degree;
   if (debugMode == DEBUG_1) cout << "Old Degree: " << oldDegree << endl;
   // Get Histogram Entires for Current and Next Degree in Degree Queue
   int valAtCurrentDegree = prepagedPageQ.getDegreeHistogramEntry(degree-1);
   if (debugMode == DEBUG_1) cout << "Histogram value at current degree: " << valAtCurrentDegree <<
endl;
   int valAtDegreePlusOne = prepagedPageQ.getDegreeHistogramEntry(degree);
   if (debugMode == DEBUG_1) cout << "Histogram value at current degree + 1: " << valAtDegreePlusOne
<< endl;
   int valAtDegreePlusTwo = prepagedPageQ.getDegreeHistogramEntry(degree+1);
   if (debugMode == DEBUG_1) cout << "Histogram value at current degree + 2: " << valAtDegreePlusTwo
<< endl;

   // Determine New Degree
   if (valAtDegreePlusTwo > 0) {
      degree = incDegree();
      degree = incDegree();
   }
   else if (valAtDegreePlusOne > 0) degree = incDegree();
   else if (valAtCurrentDegree == 0) degree = decDegree();

   if (debugMode == DEBUG_5) {
      if (degree != oldDegree) {
         cout << "New Degree: " << degree << endl;
         prepagedPageQ.printDegreeQueue(oldDegree);
         cout << endl;
      }
   }
   if (debugMode == DEBUG_1) cout << "New Degree: " << degree << endl;

   prepagedPageQ.clearDegreeHistogram();
}
unsigned int VirtualMMU::calculateBenefit(int targetAlloc)
{
   unsigned int benefit = 0;

   int index;

   for (index = 0; index < targetAlloc; ++index) {
      benefit += prepagedPageQ.getHistogramEntry(index);
   }

   return benefit;
}
unsigned int VirtualMMU::calculateCost(int targetAlloc)
{
   unsigned int cost = 0;

   int index;

   for (index = (DEFAULT_MEM_SIZE - targetAlloc); index < DEFAULT_MEM_SIZE; ++index) {
      cost += usedPageQ.getHistogramEntry(index);
   }

   return cost;
}
void VirtualMMU::initializePredictionHistograms(void)
{
   int index;
   for (index = 0; index < DEFAULT_MAX_DEGREE; index++) {
      addressLocalHistogram[index] = 0;
      recencyLocalHistogram[index] = 0;
      strideHistogram[index] = 0;
      hybridHistogram[index] = 0;
   }
}
void VirtualMMU::clearPredictionHistograms(void)
{
   int index;
   for (index = 0; index < degree; index++) {
      addressLocalHistogram[index] = 0;
      recencyLocalHistogram[index] = 0;
      strideHistogram[index] = 0;
      hybridHistogram[index] = 0;
   }
}
int VirtualMMU::calculateHistTotal(int histogram[])
{
   int total = 0;

   int index;
   for (index = 0; index < degree; index++) {
      total += histogram[index];
   }

   return total;
}
void VirtualMMU::updatePredictionHistograms(unsigned long int referencedPage)
{
   int predictionNum = 1;
   vector<unsigned long int>::iterator prediction;
   for (prediction = addressLocalPredictions.begin(); prediction != addressLocalPredictions.end();
++prediction) {
      if (predictionNum > degree) break;
      else if (*prediction == referencedPage) {
         addressLocalHistogram[predictionNum - 1] += 1;
      }
      else predictionNum++;
    }

   predictionNum = 1;
   for (prediction = recencyLocalPredictions.begin(); prediction != recencyLocalPredictions.end();
++prediction) {
      if (predictionNum > degree) break;
      else if (*prediction == referencedPage) {
         recencyLocalHistogram[predictionNum - 1] += 1;
      }
      else predictionNum++;
    }

   predictionNum = 1;
   for (prediction = stridePredictions.begin(); prediction != stridePredictions.end(); ++prediction)
{
      if (predictionNum > degree) break;
      else if (*prediction == referencedPage) {
         strideHistogram[predictionNum - 1] += 1;
      }
      else predictionNum++;
    }

   predictionNum = 1;
   for (prediction = hybridPredictions.begin(); prediction != hybridPredictions.end(); ++prediction)
{
      if (predictionNum > degree) break;
      else if (*prediction == referencedPage) {
         hybridHistogram[predictionNum - 1] += 1;
      }
      else predictionNum++;
    }
}
void VirtualMMU::printPredictionHistograms()
{
   int index;

   cout << "Address Local Prediction Histogram:\n";
   for (index = 0; index < degree; index++) {
      cout << addressLocalHistogram[index] << endl;
   }
   cout << endl;

   cout << "Recency Local Prediction Histogram:\n";
   for (index = 0; index < degree; index++) {
      cout << recencyLocalHistogram[index] << endl;
   }
   cout << endl;

   cout << "Stride Prediction Histogram:\n";
   for (index = 0; index < degree; index++) {
      cout << strideHistogram[index] << endl;
   }
   cout << endl;

   cout << "Hybrid Prediction Histogram:\n";
   for (index = 0; index < degree; index++) {
      cout << hybridHistogram[index] << endl;
   }
   cout << endl;

}
void VirtualMMU::printPredictionLists()
{
   int index;
   cout << "Address Local Prediction Histogram:\n";
   for (index = 0; index < degree; index++) {
      cout << addressLocalPredictions[index] << endl;
   }
   cout << endl;

   cout << "Recency Local Prediction Histogram:\n";
   for (index = 0; index < degree; index++) {
      cout << recencyLocalPredictions[index] << endl;
   }
   cout << endl;

   cout << "Stride Prediction Histogram:\n";
   for (index = 0; index < degree; index++) {
      cout << stridePredictions[index] << endl;
   }
   cout << endl;

   cout << "Hybrid Prediction Histogram:\n";
   for (index = 0; index < degree; index++) {
      cout << hybridPredictions[index] << endl;
   }
   cout << endl;

}
```