A Cache Memory System based on a Dynamic/Adaptive Replacement Approach

José Aguilar [*] Ernst Leiss[†]

Resumen

En este trabajo, nosotros proponemos un sistema de memoria cache basado en un esquema de reemplazo adaptativo, el cual formaría parte del Sistema Manejador de la Memoria Virtual de un Sistema Operativo. Nosotros usamos un simulador de eventos discretos para comparar nuestro enfoque con trabajos previos. Nuestro esquema de reemplazo adaptativo esta basado en varias propiedades del sistema y de las aplicaciones, para estimar/escoger la mejor política de reemplazo. Nosotros definidos un valor de prioridad de reemplazo a cada bloque de la memoria cache, según el conjunto de propiedades del sistema y de las aplicaciones, para seleccionar cual bloque eliminar. El objetivo es proveer un uso efectivo de la memoria cache y un buen rendimiento para las aplicaciones.

Palabras Claves: *Sistema de Manejo de Memoria, Memoria Cache, Evaluación de Rendimiento*.

Abstract

In this work we propose a cache memory system based on an adaptive cache replacement scheme, as part of the virtual memory system of an operating system. We use a sequential discrete-event simulator of a distributed system to compare our approach with previous work. Our adaptive cache replacement scheme is based on several criteria about the system and applications with the objective being to estimate/choose the best replacement policy. We assign a replacement priority value to each cache block according to a set of criteria to select which block to remove. The goal is to provide an effective utilization of the cache memory and good application performance.

Keywords: *Memory Management System, Cache Memory, Perfomance Evaluation.*

1. Introduction

In high-performance computer systems, memory bandwidth is often a bottleneck. There are two ways to optimize the memory access, the transformation of the code to adapt to the cache/memory system or the definition of optimal memory management systems (for example, a dynamic replacement policy). Cache memory is the simplest cost-effective way to achieve a high speed memory hierarchy. A cache provides, with high probability, instructions and data needed by the CPU at a rate that is more in line with the CPU's demand rate. Three basic cache organizations have been defined at the level of cache memory [1, 2, 4]: direct mapped, fully associative and set

[*] CEMISID. Dpto. de Computación, Facultad de Ingeniería. Universidad de los Andes. Merida - Venezuela 5101. aguilar@ula.ve

[†] Department of Computer Science. University of Houston. Houston, TX 77204-3475, USA. cosce@cs.uh.edu

associative. The choice of cache organization can have a significant impact on cache performance and cost. A cache has a fixed amount of storage. When this storage space fills, the cache must choose a set of objects (or a set of victim blocks) to evict to make room for newly requested objects/blocks. The cache replacement policy is one of the factors that determine the effectiveness of cache memories. In general, a replacement policy specifies which block should be removed when a new block must be entered into an already full cache, and should be chosen so as to ensure that blocks likely to be referenced in the near future are retained in the cache. The replacement policy's goal is to make the best use of available resources, including disk, memory space and network bandwidth. Several cache replacement mechanisms have been proposed and studied [1, 2, 3, 4, 5, 6, 8]. Despite the many replacement algorithms proposed throughout the years, approximations of Least Recently Used (LRU) replacement are predominant in actual memory management systems because of their simplicity and efficiency.

Our research attempts to improve the performance of the cache memory by developing a dynamic replacement strategy that looks at the information available (reference history, access frequency, object/block size, etc.) to make the decision what replacement technique to use, without a proportional increase in the space/time requirements. This last is accomplished through the utilization of the minimal information required for each replacement policy. In addition, we assign a replacement priority value to each cache block according to a set of criteria to select the block/object to remove. The rest of the paper is organized as follows: first, the replacement problem is presented. Then, our general model for specifying the replacement policy problem is developed. This model is used to establish a set of rules to make the decision about what replacement policy to use for each given situation. Then, the simulator is defined. Finally, some results are presented. Our model can be used by the virtual memory system of an operating system, because the information that uses our model is known by it.

2. Replacement Policy Problem

A cache provides instructions and data needed by the CPU at a rate that attempts to accommodate the CPU's demand rate. Three basic cache organizations have been defined at the level of cache memory [1, 2, 4, 8]:

- ***Direct mapped****:* A direct mapped cache maps each memory block to a unique cache block whether or not the cache block is empty. In this case, we do not require a replacement mechanism.
- ***Fully associative****:* allows a memory block to be mapped to any of the empty cache blocks, if one exist. If there is no empty cache block, a replacement policy is used to select one of the cache blocks for replacement.
- ***Set associative****:* It is a compromise between the direct-mapped and fully associative placement policies. A set-associative cache divides the cache into sets and allows a memory block to be mapped to any of the empty cache blocks within the set, if an empty cache block exists. Otherwise, similar to a fully associative cache, one of the cache blocks within the set is selected for replacement. In general, this organization offers a good balance between hit ratios and implementation cost.

The replacement policy determines how a memory block is mapped to a cache block. A replacement policy specifies which block should be removed when a new block must be entered into an already full cache; it should be chosen so as to ensure that blocks likely to be referenced in the near future are retained in the cache. The choice of replacement policy is one of the most critical cache design issues. That is, the selection of a line/block replacement algorithm, in fully

associative and set associative caches, can have a significant impact on the overall system performance. Common replacement algorithms used with such caches are [1-8]:

- ***First In-First Out (FIFO)****:* this is the simplest scheme; it is easily managed with a FIFO queue. When a replacement is necessary the first block entered at the cache memory (at the head of the queue) must be removed.
- ***Most Recent Used (MRU)****:* Replaces the block in the cache, which has been more recently used. This is not used frequently on cache memory system because it has bad temporal locality. It is a typical property of the memory reference patterns of processors, page reference patterns in virtual memory patterns, etc.
- ***Least Recently Used (LRU)****:* Replaces/evicts the block/object in the cache that has not been used for the longest period of time. The basic premise is that blocks that have been referenced in the recent past will likely be referenced again in the near future (temporal locality). This policy works well when there is a high temporal locality of references in the workload. This policy uses a program's memory access patterns to guess that the block that is least likely to be accessed in the near future is the one that has been accessed least recently. There is a variant, called Early Eviction LRU (EELRU), proposed in [5]. EELRU performs LRU replacement by default but diverges from LRU and evicts pages early when it notes that too many pages are being touched in a roughly cyclic pattern that is larger than the main memory.
- ***Least Frequently Used (LFU)****:* It is based on the frequency with which a block is accessed. LFU requires that a references count be maintained for each block in the cache. A block/object's referenced count is incremented by one with each reference to it. When a replacement is necessary, the LFU replaces/evicts the blocks/objects with the lowest reference count. The motivation for LFU and other frequency based algorithms is that the reference count can be used as an estimate of the probability of a block being referenced. In [2], Lee et al. show that there exists a spectrum of block replacement policies that subsumes both the LRU and LFU policies. The spectrum is formed according to how much more weight is given to the recent history over the older history and is referred to as the LRFU (Least Recently/Frequently Used) policy.
- ***Least Frequently Used (LFU)-Aging***: The LFU policy can suffer from cache pollution (an effect of temporal locality): if a formerly popular object becomes unpopular, it will remain in the cache for a long time, preventing other newly or slightly less popular objects from replacing it. *LFU-Aging* addresses cache pollution when it considers both a block/object's access frequency and its age in cache. One solution to this is to introduce some form of reference count "aging". The average reference count is maintained dynamically (over all blocks currently in the cache). Whenever this average counts exceeds some predetermined maximum value (a parameter to the algorithm) every reference count is reduced. There is a variant, called LFU with Dynamic Aging (LFUDA), that uses dynamic aging to accommodate shifts in the set of popular objects. It adds a cache age factor to the reference count when a new object is added to the cache or when an existing object is re-referenced. LFUDA increments the cache ages when evicting blocks/objects by setting it to the evicted object's key value. Thus, the cache age is always less than or equal to the minimum key value in the cache.
- ***Greedy Dual Size (GDS)****:* It combines temporal locality, size, and other cost information. The algorithm assigns a *cost/size* value to each cache block. In the simplest case the cost is set to 1 to maximize the hit ratio, but costs such as latency, network bandwidth can be explored. GDS assigns a key value to each object. The key is computed as the object's reference count plus the cost information divided by its size. The algorithm takes into account recency for a block by inflating the key value (*cost/size* value) for an accessed block by the least value of currently cached blocks. The *GDS-aging* version adds the cache age factor to the key factor. By adding the cache age factor, it limits the influence of previously popular documents. The

algorithm is simple to implement with a priority queue. There are several variations of the GDS algorithm that each takes into account coherency information and the expiration time of the cache. *GDSlifetime* uses the remaining lifetime of a cached object in cache as part of the key value (*lifetime/size*) to lower the priority of cached blocks about to expire. The second variation uses the observation that different types of applications change their references at different rates. *GDStype* cache replacement policy assign different key value to different types of applications (for example, HTML and text applications change more frequently; they have a high priority (*2/size*), and one uses *1* for all other applications). A last GDS variation is *GDSlatency*, which uses as key value for an object the quantity *latency/size* where latency is the measured delay for the last retrieval of the object.

- ***Frequency Based Replacement (FBR)****:* This is a hybrid replacement policy, attempting to capture the benefits of both LRU and LFU without the associated drawbacks. FBR maintains the LRU ordering of all blocks in the cache, but the replacement decision is primarily based upon the frequency count. To accomplish this, FBR divides the cache into three partitions: a new partition, a middle partition and an old partition. The new partition contains the most recent used blocks (MRU) and the old partition the LRU blocks. The middle section consists of those blocks not in either the new or old section. When a reference occurs to a block in the new section, its reference count is not incremented. References to the middle and old sections do cause the reference counts to be incremented. When a block must be chosen for replacement, FBR chooses the block with the lowest reference count, but only among those blocks that are in the old section.
- ***Random (RAND)****:* It chooses among all blocks in the cache with equal probability. Intuitively, RAND seems appealing in this context if the clients caches are filtering out all of the locality characteristics from their reference streams. RAND provides a kind of lower bound on performance. That is, there is no reason to use any policy that performs worse than RAND. Thus, this policy must be used if it is the faster and less expensive.
- ***Priority Cache (PC)****:* Uses both runtime and compile-time information to select a block for replacement. PC associates a data priority bit with each cache block. The compiler, through two additional bits associated with each memory access instruction, assigns priorities. These two bits indicate whether the data priority bit should be set as well as the priority of the block, i.e. low or high. The cache block with the lowest priority is the one to be replaced.
- ***Prediction****:* An optimal cache replacement policy would know a document's future popularity and choose the most advantageous way to use its finite space. Unfortunately this requires future knowledge, and even with perfect future knowledge it is still computationally expensive.

In general, the policies anticipate future memory references by looking at the past behavior of the programs (program's memory access patterns). Their job is to identify a line/block (containing memory references) which should be thrown away in order to make room for the newly referenced line that experienced a miss in the cache. Relative performance of these algorithms depends mainly on the length of the history consulted, but they ignore the cache block state information that is also indicative of the program's characteristics. There are several criteria to measure the performance of a cache replacement technique:

- ***Miss rate****:* number of miss references. It is the most popular measure of cache efficiency. A rate of 70 percent indicates that seven of every ten requests to the cache found the requested object.
- ***Object Retrieval time****:* it is especially of interest to end users. Latency is inversely proportional to object hit rate because a cache hit can be served more quickly than a request that must pass through the cache to an origin server and back.

- ***CPU, I/O system and network utilization***: the fraction of total available CPU cycles or disk or memory or network bandwidth consumed by the replacement technique (update time overhead and execution time overhead).
- ***Stale ratio***: the number of the stale blocks/objects on the cache system.
- ***Byte hit rate***: the number of bytes returned directly from the cache as a fraction of the total bytes accessed. This measure is not often used in system architecture cache studies because the objects (cache lines) are of constant size, and therefore the byte hit rate is directly proportional to the object hit rate. It is interesting on the Internet because external network bandwidth is a limited and often expensive resource. A byte hit rate of 30 percent indicates that three of every ten bytes requested were returned from the cache, while 70 percent of all bytes returned to users were retrieved across the external network.

3. Our adaptive replacement policy

We propose an alternative dynamic cache management policy that chooses among different replacement strategies depending upon different criteria. In addition, we assign a replacement priority value (key value) to each cache block according to a set of criteria to select the block to replace. A general procedure is:

1. Determine the reference locality on the cache memory (replacement phase).
2. Update variables of the cache memory and replacement systems.

In our approach, the replacement phase can be formulated according to the next algorithm:

1. If *reference miss and cache full* then
   1.1 Choose replacement mechanism.
   1.2 Apply replacement and coherence mechanisms.
   1.3 Update variables of the cache memory and replacement systems.
2 If *reference hit* then
   2.1 Update variables of the cache memory and replacement systems.
3 If *cache not full and reference miss* then
   3.1 Assign new block on the free cache space
   3.2 Update variables of the cache memory and replacement systems.

3.1 Replacement Criteria

Typically, a cache replacement technique must be evaluated with respect to a given workload that describes the characteristics of the requests to the cache. Of particular interest are the patterns in the number of objects referenced and the relationships among accesses. Workload is sufficiently complicated that we can use other types of information to try to solve this problem. Thus, we define a set of parameters that we can use to select the best replacement policy in a dynamic environment:

A) Characteristics of the system
- Type of cache memory
  + Direct mapped cache
  + Full-associative
  + Set-associative
  + Local/private or distributed cache memory

- Information about the system
  + Workload
  + Bandwidth
  + Latency
  + CPU Utilization
- Type of system
  + Shared memory with local cache
  + Distributed system with local cache

B) Information about the application
- Information about the data and cache block or objects
  + Frequency
  + Age
  + Size
  + Access Pattern
- Type of application
  + Spatial Locality (SL)
  + Temporal Locality (TL)

C) Other information
  + Loop Transformation
  + Cache conflict resolution mechanism
  + Pre-fetching mechanism

3.2 Our Model

An optimal cache replacement policy would know the future workload. In the real world, we must develop heuristics to approximate the ideal behavior. For each policy, we define the information required:

- LFU: reference count.
- LRU: the program's memory access patterns.
- Priority Cache: information at runtime or compile time (data priority bit by cache/block).
- Prediction: the entire program's memory access pattern but with reduced time/space requirement.
- FBR: the program's memory access patterns and organization of the cache memory.
- MRU: the program's memory access patterns.
- FIFO: the program's memory access patterns.
- GDS: size of the objects, information to calculate the cost function, reference count.
- Aging approaches:
  + GDS-aging: GDS age factor.
  + LFU-aging: LFU age factor.

We define one expression, called the *key value*, to define the priority of replacement of each block/object. According to this value, the system chooses the block with higher priority to replace (low key value). The key value is defined as:

Key-Value = (CF+A+FC)/S + cache factor
(1)

where - FC is the frequency/reference count, that is the number of times that a block has been referenced,
- A is the age factor,
- S is the size of the block/object,
- CF is the cost function that can include costs such as latency or network bandwidth.

The first part of Equation (1) is typical for the GDS, LRU and LFU policies (using information about objects to reference and not about cache blocks). Cache factor is defined according to the replacement policy used:

- LFU: blocks with a high frequency count have the highest cache factor.
- LRU: least recently used blocks have the highest cache factor.
- Priority Cache: defined at runtime or compile-time.
- Prediction: least used block in the future has the highest cache factor.
- FBR: least recently used block has the highest cache factor.
- MRU: most recently used block has the highest cache factor.
- FIFO: block at the head of the queue has the highest cache factor.
- GDS: not applicable.
- Aging approaches: FC/A, with a reset factor that restarts this value after a given number of ages or when the age average is more than a given value.

According to this, our *General Adaptive Algorithm* is the following:

1. If *reference miss and cache full* then
   1.1 Choose replacement mechanism according to a *decision system*.
   1.2 Calculate the cache factor for each block.
   1.3 Run the replacement mechanism.
   1.4 Update variables of the cache memory and replacement systems (FC, A, CF).
2. If *reference hit* then
   2.1 Update variables of the cache memory and replacement systems (FC, A, CF).
3. If *cache not full and reference miss* then
   3.1 Assign new block on the free cache space.
   3.2 Update variables of the cache memory and replacement systems (FC=0, A=0).

The update of the variables can be different depending on whether the system has a write miss or read miss. The *Decision System* is composed of a set of rules to decide the replacement policy to use. Each rule selects a replacement policy to apply according to different criteria:

If *TL is high and the program's memory access pattern is regular* then
    Use a *LRU* replacement policy
If *TL is low and the program's memory access pattern is regular* then
    Use a *LFU* replacement policy
If *TL is low and SL high* then
    Use a *MRU* replacement policy
If *we require a precise decision using a large program's memory access pattern* then
    Use a *Prediction* replacement policy
If *objects/blocks have variable sizes* then
    Use a *GDS* replacement policy
If *a fast decision is required* then
    Use the faster and less expensive replacement policy (for example, the *RAND* replacement policy).

If *there is a large number of LRU blocks* (several blocks least recently used are candidates to be replaced*)* then
    Use a *FBR* replacement policy
If *SL is high* then
    Use a hybrid *FBR* + *GDS* replacement policy
If *the program's memory access pattern is irregular* then
    Use an *Aging* replacement policy
If *the CPU utilization is low* then
    Choose a process to suspend (In this case, we can use a *PC* policy to select the process)
    Reinvoke the *General Adaptive Algorithm*

For the rest of the cases, choose randomly a replacement policy. The CPU utilization criterion avoids the starvation or thrashing problem on the system. When its utilization is low, that can due to processes that spending more time paging than executing (they need more cache space to avoid quick miss references), or processes that are waiting indefinitely (we must recovery the system from a deadlock, for example, suspending one process at a time until the deadlock cycle is eliminated). That is the reason to suspend a process. In general, these rules are based on the average of the different criteria on the system. We can take into account the victim's information (the process that has been selected to expel its page) with the next rule:

If *the victim's SL is high or the victim's TL is high* then
    Lock this page % Choose another page
    Reinvoke the *Decision System*

If we don't find a page to expel, we choose the first victim that we had selected and we suspend this process.

4. Simulator

In this section, a discrete-event simulator is presented to simulate a cache memory system based on the previous replacement mechanism. The cache parameters to be used in this simulator will be cache size, associativity, number of frames per cache, and cache operation to be executed (e.g., cache read or cache write). A Process is identified per an IP (Identification Process) and defined by a page reference vector such as:

1 1 1 2 3 3 3 4 4 4 ...

where each element *i* represent the page that the process references at the i^{th} time. Basically, in our system a memory address submitted to the cache will be divided into three fields: a IP, a page, and an offset. The general procedure is the following:

- Generate randomly an initial process with its arrival time and page reference vector.
- Make an initial cache allocation.
- Emulate the behavior of the cache system using the simulator. During the simulator execution time, our cache replacement mechanism is called, depending on the utilization of the cache memory.
- Calculate performance measures (cache hit rate, occupation time of processors, program execution times, etc.).

The simulation is carried out managing a future event list (FEL). Each element of the FEL has the form: (occurrence time of the event, event type). Let t_i be the occurrence time of event *i* and e_i be the type of event *i*. The algorithm of the simulation is the following:

- Take the next event (t_i, e_i) of the FEL
- Until clock=total simulation time or accuracy for confidence intervals is reached
  - Clock= t_i
  - Execute procedure e_i for event
- Collect statistics

One simulation run corresponds to the execution of one set of processes using one set of parameters (size of the cache memory, number of frame, process arrival times, etc.). We have defined the following events:

1. ***Arrival of a process i****:* In this event, a process can start its execution if the system can assign it the minimal number of pages that it needs to execute (*n*). The time between two successive arrives is exponentially distributed with parameter α. In this event, the corresponding procedure does the following:

   - Determine if we can assign the minimal number of pages that the new process *i* needs to execute (n_i). In this way we avoid the thrashing problem because we guarantee that each process into the ready queue has allocated its minimal number of pages.
   - Generate event (end of process *i*, actual value of clock + number of elements on the reference vector) if process *i* can be executed.
   - Generate event (end of process *i*, actual value of clock + infinite) if process *i* can't be executed.
   - Assign the minimal number of pages (n_i) + *extra* to start the execution of the process *i* if it can be executed and the memory is not full. Otherwise, send process to the waiting queue. *extra* is an extra number of pages that we assign to a given new process.
   - Call our replacement mechanism if it can be executed and the memory is full. Otherwise, send process to the waiting queue.
   - Determine process *j* whose pages have been expelled.
   - Generate event (page fault from process *i*, actual value of clock + next fault reference) according to the reference vector of the process *i* if process *i* is not in the waiting queue.
   - Generate event (page fault from process *j,* actual value of clock + next fault reference) according to the reference vector of the process *j* if process *j* has been expelled.
   - Update system variables.
   - Generate Tj = time to arrive a new process *j* according to a Poisson process with parameter α.
   - Generate event (arrival of a process *j*, actual value of clock + Tj).

2. ***End of process i****:* When a processor *i* finishes its execution, the system verifies if one of the processes in the waiting queue can start its execution. A process finishes its execution when we arrive at the end of its reference vector. In this event, the corresponding procedure does the following:

   - Release frames assigned to process *i*.
   - Update system variables.

- Get a new process *j* from the waiting queue (if there exists one in the waiting queue).
- Generate event (restart a process *j*, actual value of clock + 1).

3. ***Page fault from process i****:* This event is executed when there is a missing reference in the process *i*. We need to verify if the system has a thrashing problem. In this case, we must suspend the process *i.* In this event, the corresponding procedure does the following:

   - Determine if we don't have a thrashing problem (that means, if we can assign the minimal number the pages that the process *i* needs to continue).
   - Update event (end of process *i*, actual value of clock + number of elements not yet reference on the reference vector) if process *i* can be executed.
   - Update event (end of process *i*, actual value of clock + infinite) if process *i* can't be executed.
   - Assign the referenced page to the memory if process *i* can be executed and the memory is not full. Otherwise, send process to the waiting queue, save its current state and release the frames assign to it.
   - Call our replacement mechanism if process *i* can be executed and the memory is full. Otherwise, send process *i* to the waiting queue, save its current state and release the frames assign to it.
   - Determine process *j* whose pages have been expelled.
   - Generate event (page fault from process *i*, actual value of clock + next fault reference) according to its reference vector if process *i* can be executed.
   - Generate event (page fault from process *j,* actual value of clock + next fault reference) according to the reference vector of the process *j* if process *j* has been expelled.

4. ***Restart a process i****:* This event is executed when we restart the execution of an event that is in the waiting queue. In this event, the corresponding procedure does the following:

   - Determine if we can assign the minimal number the pages that the process *i* needs to execute.
   - Update event (end of process *i*, actual value of clock + number of elements not yet referenced on the reference vector) if process *i* can be executed.
   - Update event (end of process *i*, actual value of clock + infinite) if process *i* can't be executed.
   - Assign the minimal number of pages + *extra* to start its execution and delete process *i* from the waiting queue if it can be executed and the memory is not full. Otherwise, keep process *i* on the waiting queue.
   - Call our replacement mechanism and delete process *i* from the waiting queue if it can be executed and the memory is full. Otherwise, keep process *i* on the waiting queue.
   - Determine process *j* whose pages have been expelled.
   - Generate event (page fault from process *i*, actual value of clock + next fault reference) according to its reference vector if process *i* can restart its executed.
   - Generate event (page fault from process *j,* actual value of clock + next fault reference) according to the reference vector of the process *j* if process *j* has been expelled.

4.1 Simulations

We will take into account the following parameters:

- Number of Frames (Cache size) [32-2048]: A cache is composed of a number of frames. In our paper, the frame size is equal to the page size.
- Process Page Average Number [8-128]: This parameter represents the number of pages that a given process will request during its execution. Also, a process *i* is described by the minimal number of pages that it needs to be executed (n_i). In our paper, this number is one half of the number of pages that each process requests.
- Associativity [1-1024]: This parameter determines the number of frames per set in a cache. When *Associativity = 1*, the cache memory is a *direct mapped* cache. When *Associativity = Cache size*, the cache memory is a *fully associative* cache. In other cases (*1 < Associativity < Cache size*), the cache memory is a *set associative* cache and the number of frames per set is defined by the value of Associativity.
- Mean Arrival Time (α) [1-5]: The time between two successive arrives is exponentially distributed with parameter α.

We compute the following performance measures to evaluate the effectiveness of the fault-tolerant protocol:

- Replacement Mechanism Overhead,
- Miss Ratio,
- Process Mean Waiting Time.

The standard case uses the following parameter values: Number of Frames: 1024, Process Page Average Number: 128, Associativity: number of frames, Process Mean Arrival Time: 3. For obtaining each point in our tables, we have carried out 30 simulations for each problem instance. We use as input to our simulator, address traces from two SPEC92 benchmark programs (Nasa7 and Alvinn) [4]. We compare our approach with the PPN (probabilistic neural network) predictive mechanism based on Artificial Neural Network proposed in [4] and the LRU conventional mechanism proposed in [7] (it is one of the best performing cache replacement policy referenced in the work about cache replacement policies).

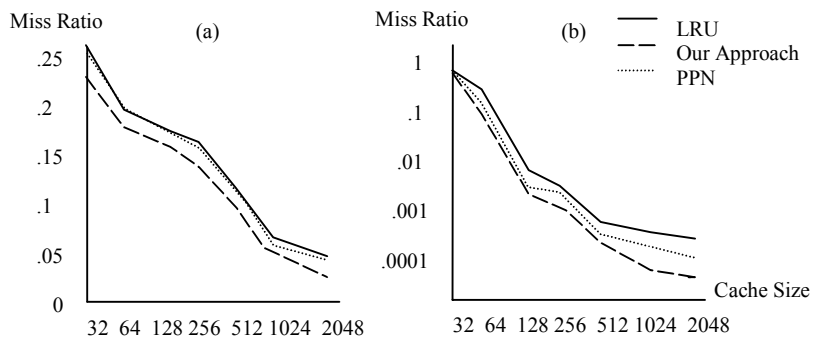

Figure 1. Miss Ratio versus Cache Size with Nasa7 (a) and Alvinn (b)

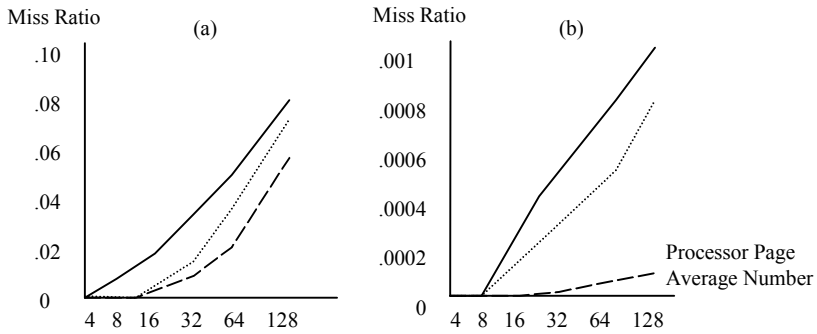

Figure 2. Miss Ratio versus Process Page Average Number with Nasa7 (a) and Alvinn (b)

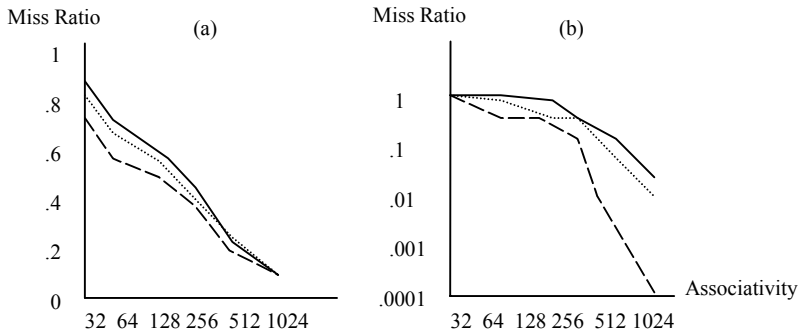

Figure 3. Miss Ratio versus Associativity with Nasa7 (a) and Alvinn (b)

Figures 1, 2, 3 and 4 illustrate that the best performance of our approach occurs for the different values of the parameters, but figures 5 and 6 show that the overhead of our approach is very important, especially when the system size is small (number of pages ≤ 128) or when the system is overloaded (process mean arrival time ≤ 2). That is due to the decision system module of our mechanism whose execution time must be reduced for these cases (one way to reduce this time is to preselect a given replacement policy when the system dimension is small). In the case of the figure 3, the cache memory is a *set associative* cache when Associativity < 1024, and the cache memory is a *fully associative* cache when Associativity = 1024.

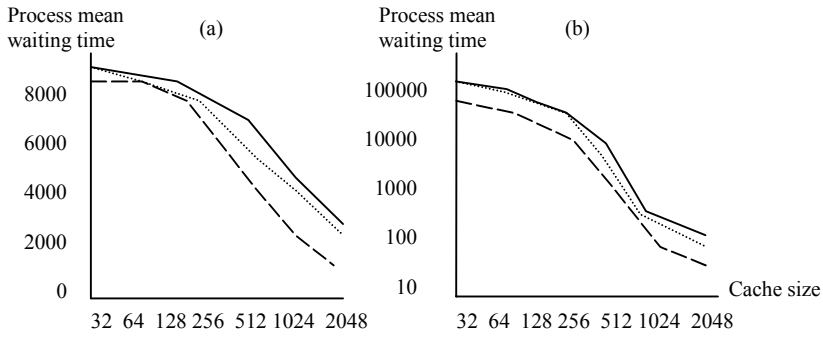

Figure 4. Process mean waiting time versus cache size with Nasa7 (a) and Alvinn (b)

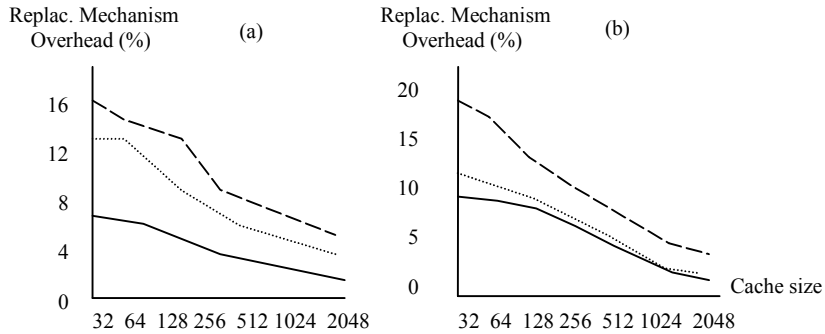

Figure 5. Replacement mechanism overhead versus cache size with Nasa7 (a) and Alvinn (b)

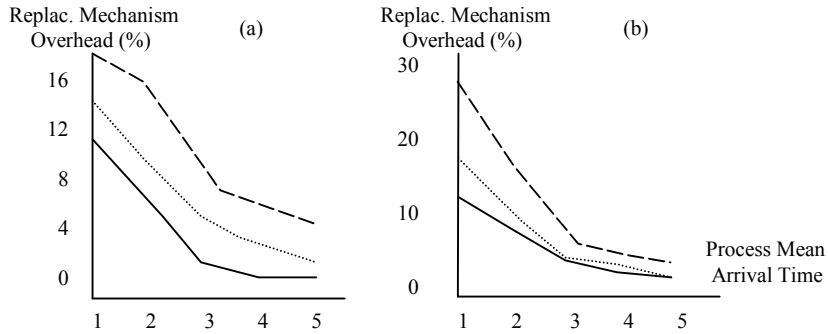

Figure 6. Replacement mechanism overhead versus Process mean arrival time with Nasa7 (a) and Alvinn (b)

In general, our approach reduces the process mean waiting time, increasing the CPU utilization. This is a way to improve the process execution time. The performance of our approach depends heavily on the size of the system.

5. Conclusion

The goal of this research was to formulate an overarching framework subsuming various cache management strategies in the context of different situations. Our approach includes additional information/factors such as frequency of page use, in replacement decisions. Our approach opens a new direction for further research in managing disk caching, distributed caching, and page replacement in virtual memory. Our model obtains good performance, but incurs a relatively high overhead when the dimension of the system is small. Our system works very well when the system is large because the overhead of our system is compensated by the improvement at the level of the utilization of the cache memory and the process execution time. We are extending this model for the case of web caching and distributed caching. We are planning to incorporate coherency issues in the cache replacement policy.

Acknowledgment

This work was partially supported by CDCHT-ULA grant I-620-98-02-AA. Jose Aguilar was supported by a CONICIT-Venezuela grant (subprograma de pasantías postdoctorales).

References

[1] Cho S., King J., Lee G., "*Coherence and Replacement Protocol of DICE-A Bus Based COMA Multiprocessor*", Journal of Parallel and Distributed Computing, Vol. 57, pp. 14-32, 1999.

[2] Lee D., Choi J., Noh S., Cho Y., Kim J., Kim C., "*On the Existence of a Spectrum of Policies that Subsumes the Least Recently Used (LRU) and Least Frequently Used (LFU) Policies*", Performance Evaluation Review, Vol. 27, N, 1, pp. 134-143, January 1999.

[3] Mounes F., Lilja D., "*The Effect of Using State-based Priority Information in a Shared-memory Multiprocessor Cache Replacement Policy*", IEEE Computer, Vol. 2, pp. 217-224, 1998

[4] Obaidat M., Khalid H., "*Estimating NN-Based Algorithm for Adaptive Cache Replacement*", IEEE Transaction on System, Man and Cybernetic, Vol. 28, N. 4, pp. 602-611, 1998.

[5] Smaradakis Y., Kaplan S., Wilson P., *"EELRU: Simple and Effective Adaptive Page Replacement"*, Performance Evaluation Review, Vol. 27, N, 1, pp. 122-133, January 1999.

[6] Tyson G., Fonrens M., Matthews J., Pleczkun A., *"Managing Data Caches Using Selective Cache Lien Replacement*", International Journal of Parallel Programming, Vol. 25, N. 3, pp. 213-242, 1997.

[7] Puzak T., "Analysis of Cache Replacement Algorithms", Ph.D dissertation, Dept. Elect. Compt. Eng., Univ. Mass, Boston, 1985.

[8] Aguilar J., Leiss E., "A Proposal for a Consistent Framework of Dynamic/Adaptive Policies for Cache Memory Management", Technical Report, Department of Computer Sciences, University of Houston, April 2000.